Understanding Significance Testing as the Quantification of Contradiction

Conceptual and Practical Foundations of Statistics in the Context of Experimental Research

Thomas E. Gladwin

1. Introduction

This text is meant to help strengthen statistical understanding, leaning strongly on the use of programming simulations rather than formal mathematics. It's not an introductory textbook, but more something to (re-)read alongside your usual textbook during or after undergraduate modules. I'm not going to explain how to calculate a mean or a variance, for instance. Nevertheless, in a sense it is a "pre-statistics" book, in that it's about very basic conceptual foundations. You'll also learn a tiny bit about data analysis and simulation using Python.

I'll conclude by arguing that null hypothesis significance testing is a perfectly valid method for experimental research.

A tiny bit of motivating philosophy of science to start. Perhaps you feel that "truth" is a hopelessly outmoded and naive concept; however, I would posit that it's very clear that lies do exist; stupidity exists; gaslighting exists; mistakes exist; and biases exist. If so, the opposite of those things exists, which we might as well call truth: the things that lies and errors take us away from. When we do a scientific experiment, we want to move towards the truth about something. And we need statistics for that to work. Oh really, I hear a chorus of smug pseudoscientists ask, why can't we just look and think really deeply and use our intuition and personal wisdom and insight? Numbers are so superficial! Look, we can make up loads of non-statistical rules to follow completely rigorously, and they can involve quite onerous activities, so why aren't those just as valid a method? And aren't you silly to think you can be perfectly 100% objective? This is a disappointingly common and quite damaging mix of nonsense, ignorance, and strawman arguments that learning statistics will help you see through. The reason that statistics was developed was precisely because it's necessary to prevent us from making fools of ourselves. ***Statistics starts by embracing the humbling awareness that we as humans are absolutely untrustworthy when it comes to interpreting data***. We recognize patterns in complete noise, unless you believe that cloud over there actually is a giant shark or there actually are faces looking at you from a huge range of surfaces involving swirly patterns. What a terrifying life you must lead! And of course, we have a history of bitter disagreements between people who have different interpretations of what they see. So, if you want to base science on subjectivity, did you mean your pure and insightful subjectivity, or that garbage professor Johnson's depraved and moronic subjectivity? Should I then add in my own subjective perspective to decide which of your subjective perspectives is better? But maybe your subjective perspective on which of those subjective perspectives differs from mine… I prefer the alternative: Let's make a good-faith effort to find ways to measure things as objectively as we can, in a real world we at least agree exists, and draw conclusions that tend to be valid, i.e., that will move us towards truth about that real world in the long term. We'll still be wrong, a lot, but if we work to falsify beliefs that are untrue then at least we can hope that by elimination we'll tend to retain beliefs that are true. If you care about that, then statistics is for you.

A practical reason to care about the kind of objective truth that goes hand in hand with quantification (Wigner, 1960) is that it includes things like:

- Do people die less with medicine X or medicine Y?
- Does wearing face masks slow the spread of a virus?

- Do people with depression have better outcomes with one form of therapy or another?
- Do children with certain behavioural problems have a higher risk of ending up in prison?

We want to know these things. But more subtle things are similarly better understood, in more depth, if we can dive beneath the superficial surface of common sense and personal subjectivity. We can ask completely non-intuitive questions, such as whether oscillations of specific frequencies in the brain, evoked by very particular psychological tasks, can predict whether a teenager is likely to escalate their drinking. We could model complex relationships between implicit measures to identify changes in self-identity networks that predict recidivism. We could try to understand the reciprocal interactions between neuromaturation, parenting style, and social environment that result in strong self-regulatory abilities. And so on – we will test ideas about things we would never had considered – would never have been able to conceive - if we hadn't gone down the path of quantification and statistics.

Statistics is difficult to either learn or teach well. This text aims to play a small role in helping with that, but as a student it's important to realize that some things just need to be learned differently. Statistics requires a consistent attention to detail, rigour, effort, and perseverance that you can get away with not applying for other material in, say, a psychology course. The upside is that you're actually learning something; you're building up a really valuable system of knowledge and skills. And what you need to have faith in at first, and then discover, is that it all makes sense, even things that seemed totally impenetrable at first. But this can require not only personal effort but also finding that one source that explains it just right. Be aware that this may well not be a textbook – it could be an article, a blog, or a video. Whatever works. Importantly, with the programming patterns you'll learn in this book you'll be able to test your understanding via simulations.

2. Inference: From sample to population

There are lots of ways we can fall short in our attempt to know the truth, and all these ways generate uncertainty. The most basic type of uncertainty, the thing that's absolutely essential for understanding basic statistics, is something called ***sampling error***. This is what happens when we ***want*** to know the truth about a so-called ***population***, but we only have a ***sample*** of actually collected data to draw inferences from. For instance, the "sample" could be the questionnaire scores from a group of 175 people who participated in a study. The "population", in contrast, is everyone who ***could have*** gone into the sample; it's the ***total potential pool*** of the type of participants that the sample was drawn from. For instance, if we randomly select 50 university lecturers to fill in a questionnaire on moral injury, our sample is those 50 specific people, and our population is all university lecturers. Or maybe we can be more precise: if the lecturers are all from psychology departments, then the actual population is psychology lecturers. What population we can even try to draw inferences about depends on the population we actually drew the sample from.

Our problem is the gap between the specific sample and the larger population – we have the data of the sample, but we want to ***infer*** something about the population. Every time we draw a different sample from a population, that sample could lead us to draw a different conclusion. For instance, we may urgently need to know the average airspeed velocity of an unladen swallow. If we cannot avoid the question, we could capture a swallow and measure how fast is flies. OK. But maybe we captured a particularly slow or fast bird. Well, let's look at a different one. Oh, this one is faster! Well, we can take their average. But that average would be different if we'd captured two different swallows… We want to know the true average of all unladen swallows in general, not just the two we happen to have in hand!

We're going to be using Python for hands-on exercises (or use a different language like R if you prefer and are happy to translate the code). Please actually do the programming to build skills and avoid only a very superficial grasp of the material. Install Python and, if necessary, follow a quick basic tutorial on Python programming. I'd suggest using the Anaconda distribution for Windows, and working in the Anaconda prompt, using a text editor to make scripts.

Let's ***simulate*** the problem of the airspeed velocity of the unladen swallow. Simulations are the poor man's version of mathematic sophistication, to paraphrase my PhD supervisor; they're incredibly helpful to understand things but also to test and calculate things in practice. We're going to play God and create a population of swallows, and, being, omniscient, we now know the truth: the airspeed is 11 m/s. But we're going to simulate the capture of random pairs from that population – we'll look at what happens in 10 different captures. Put this code in a Python script (e.g., swallows.py) and run it (`python swallows.py` in the Anaconda prompt) to create the population; ***do this before continuing***:

```
# Install the numpy toolkit for scientific computing, which has functions for doing data
analysis

import numpy as np


# Create the population of faster and slow swallow

# We'll first create a population generated by sampling from a standard normal distribution,
using randn(). Note this special case of a sampling procedure is to make our population; we'll
be drawing "samples" in a statistical sense from our population.

# Then we'll set the population-mean to exactly 11, representing 11 m/s. First we remove the
mean we got after the first random generation, so we know the mean must be exactly zero at
that point; then we add 11 to all the values.

# Finally we'll output the population to check what's going on.

Total_number_of_swallows = 50

Speeds_Population = np.random.randn(Total_number_of_swallows)

Speeds_Population = Speeds_Population - np.mean(Speeds_Population)

Speeds_Population = Speeds_Population + 11

print(Speeds_Population)

print(np.mean(Speeds_Population))
```

So now we've played God and have a population (abstracted to an array of airspeeds, one per swallow) with an average airspeed velocity of 11 m/s. But now let's take the role of non-gods who don't have omniscience; but who can catch birds. Add this code to represent a random capture of a pair of birds:

```
# Now we're going to draw a random sample of two swallows.

My_Sample = np.random.choice(Speeds_Population, size=2, replace=False)

print('The captured swallows had speeds:')

print(np.round(My_Sample, 2))

print('The average speed is this sample was:')

print(np.round(np.mean(My_Sample), 2))
```

Run the program a few times and note how every time you pick a new random sample, the sample-mean changes – but the true mean, i.e., the mean of the population, is set to exactly 11 m/s! ***This fluctuation of the sample-mean around the population-mean is sampling error***. And it means that, if you just look at the sample-mean and say, hah, now I know the population-mean, you're almost always going to be wrong.

But statistics will help us be ***less*** wrong over time. It does this by letting us know how likely – or more to the point, how unlikely – it is to find samples that look very different from the population someone is claiming we're sampling from. If someone horrible comes up and claims that the average airspeed velocity of the unladen swallow is 20 m/s, maybe all I need to do to satisfy myself is show everyone what a liar that person is. Or maybe someone tells me that the average airspeed of the laden swallow is lower rather than that of the unladen swallow, and all I want to do is check whether that person really has any evidence for that claim. I might not care about the actual airspeed – just about arguments involving contradictions between claims and available evidence. That's what I'll try to show is what statistical significance testing is all about.

3. Very basic probability: Infinite imaginary worlds

All of statistics involves uncertainty and probability. If you dislike the concept of uncertainty, get over it quickly – we can make ***good faith efforts*** to be ***as objective as possible*** and to aim at ***approaching truth***, but we're not an omniscient God except when we're running simulations. But what is probability?

People disagree about how to conceive of probability. One way is called "frequentism" and is sadly often misunderstood and/or misrepresented by its critics. The essential thing to understand about frequentism is that it's all about infinite imaginary worlds, or the multiverse if you prefer (Deutsch, 1997). What we do in frequentism is ***consider all the relevant outcomes that could have happened*** and define the probability of a certain event as the number of outcomes that fall within that event, divided by the number of possible outcomes. An event, for instance, could be that a die throw gives a value of 3 or lower; the outcomes belonging to that event would be throwing a 1, 2, or 3; and there are 6 possible outcomes; so the probability of the event would be 3/6 = 0.5. Or if we throw two dice, there are 36 outcomes, and we might want to know the probability of the event "the sum of the two dice is 5". We would visualize the outcome space as below:

		Die 1					
		1	**2**	**3**	**4**	**5**	**6**
Die 2	**1**	2	3	4	5	6	7
	2	3	4	5	6	7	8
	3	4	5	6	7	8	9
	4	5	6	7	8	9	10
	5	6	7	8	9	10	11
	6	7	8	9	10	11	12

The outcomes that are true for the event "the sum of the two dice is 5" are shaded below:

		Die 1					
		1	**2**	**3**	**4**	**5**	**6**
Die 2	**1**	2	3	4	5	6	7
	2	3	4	5	6	7	8
	3	4	5	6	7	8	9
	4	5	6	7	8	9	10
	5	6	7	8	9	10	11
	6	7	8	9	10	11	12

We have 4 out of 36 outcomes in the event, and the probability of the event is 4/36. We expect that if we throw two dice a million times, we'd get about 111111 of those throws to have summed to a 5.

We could also have an event about range of values. For instance, what's the chance of throwing at most a three or at least an eleven?

		Die 1					
		1	**2**	**3**	**4**	**5**	**6**
Die 2	**1**	2	3	4	5	6	7
	2	3	4	5	6	7	8
	3	4	5	6	7	8	9
	4	5	6	7	8	9	10
	5	6	7	8	9	10	11
	6	7	8	9	10	11	12

We have a probability of that of 6/36 = 1/6.

Try it for yourself by sketching the table like above: What's the probability of the sum being at most three? What's the probability of Die 1 being exactly one higher than Die 2? What's the probability of Die 1 and Die 2 being at least three values apart (so, e.g., "Die 1 is 1 and Die 2 is 4" would be an outcome that belongs to the event, as would "Die 1 is 6 and Die 2 is 3")?

One of the strawmen arguments used against frequentism is that it can't be used when outcomes couldn't be repeated many times in reality. It's hopefully clear that this is silly, since we're talking about frequencies in models of abstract, conceptual, counterfactual outcomes – every time we think in terms of frequentist probability, we open an imaginary multiverse of infinite worlds! This many-worlds definition of probability is an incredibly adaptable way to define probability, in combination with the axiomatic definitions from mathematics. This isn't covered here in detail, but it's highly recommended you learn enough maths for that. The basic idea is that we define probability in terms of the rules of how it has to behave mathematically to represent our intuitive and empirical understanding of probability. These are the three Kolmogorov axioms: The probability of an event is a non-negative real number; the probability that at least one of the outcomes will occur is 1; the probability of the combination of mutually exclusive events is the sum of the events' probability. What we model is all the possible outcomes that could occur – and remember, a model is always a simplification of only certain elements of reality that we care about. If we want to know the chance that a precious, unique vase falls off a wobbly pedestal that a cute puppy bumps into, we would find a way to model all the possible ways the pedestal ***could*** tip, within our model. In reality, the precious vase will obviously quite likely not fall and smash more than once. But we can still count frequencies of outcomes from our model's conceptual outcome space and see how many would fall under a certain event.

In statistics, our "outcomes" are generally the scores we get in a randomly selected sample. The probability of an event involves the proportion of outcomes, over all possible samples we could have drawn according to a certain statistical model, with a certain feature defining an event. That event could for instance be having a sample-mean above a certain critical value (we'll go into critical values later). We might want to know how likely it really is to find a reaction time difference on an attentional bias task of +100 ms is there's actually no effect of the cues. Or we might want to know whether the apparent decrease in anxiety symptoms in an intervention, versus a control group, could be explained by random fluctuations.

We generally aren't going to need to do the maths that go into statistical techniques, but mostly for the sake of tradition we do need to understand a general point: the probability of a certain kind of event often depends on ***parameters*** called "degrees of freedom" or *df*. This parameter generally has to do with how many random values are being added together to calculate a sample-mean (or a sample-statistic in general – a "statistic" is any score that we calculate using data from a random sample). As we'll see later, the *df* affect the kinds of sample-events you're likely to get and therefore what inferences you can draw from an observation.

4. Does a sample-mean "significantly" differ from a hypothesized value?

In this section I'll introduce the core concept of "significance" and "significance testing". This is all about thinking about ***sampling error***.

We're going to expand the swallow simulations. Instead of just taking one sample – i.e., capturing a pair of swallows once – we're going to make the code simulate taking 100 samples. We're just going to put the random sampling in a loop, as follows; note that I've put the number of birds we're going to capture ***in each separate sample*** in a variable. ***Run*** the code below:

```
# Now we're going to simulate what happens when we take a lot of different samples.
Number_of_samples = 100
Birds_per_sample = 2
for this_sample in range(Number_of_samples):
    My_Sample = np.random.choice(Speeds_Population, size=Birds_per_sample, replace=False)
    print(np.round(np.mean(My_Sample), 2))
```

Look at the numbers this generated: each number represents the sample-mean (the average of the two birds you captured) of one of the samples you took. My output was as below; the square brackets surrounding the list of numbers mean that we're looking at an array of numbers:

Simulated sample-means:

[11.15 9.85 10.87 10.83 10.43 10.8 11.68 10.76 10.22 10.37 11.51 10.27
9.95 11.49 11.95 10.51 10.79 11.54 10.2 11.16 10.72 12.4 10.23 12.36
10.6 10.39 12.11 10.62 10.61 10.29 10.07 10.87 10.96 11.47 11.96 10.52
11.61 12.22 12.19 10.36 11.17 10.45 10.82 12.03 10.94 10.97 10.42 11.25
11.44 10.82 11.48 10.08 10.93 11.21 11.07 11.39 11.11 12.52 12.04 11.25
11.34 10.31 10.67 11.9 10.18 10.19 11.9 11.94 12.   10.28 10.91 10.75
10.84 11.95 11.3 10.8 11.44 11.35 10.85 11.35 9.42 10.97 11.88 11.14
10.97 11.13 10.69 10.86 10.88 11.13 10.52 11.65 12.29 10.85 11.14 11.87
10.73 9.91 11.27 11.71]

Notice how the sample-means vary! What range do the value have, roughly, what minimum and maximum do you see? In my simulations, I'm seeing sample-means vary very roughly between 9 and 12.

Note down the last sample-mean you get. For me that was 11.71. We're going to use that later, as if this was the value we got doing real-life data collection.

Now, in our simulations we're an omniscient God, but as scientists we only have a particular sample to work with (this can lead to bitterness and is why some scientists are atheists). Because of sampling error, we can't just look at our ***sample-mean*** and say, hah, this is the ***population-mean***, and pretend we now indeed know the average airspeed velocity of a swallow. What we can do, however, is ***test hypotheses*** about the population-mean. We can check whether a certain belief is ***contradicted*** by the observed data. This is pretty much a concrete way of being a Popperian and making scientific progress via falsification (Popper, 1934): We make a guess about the population and then take samples to check whether what we observe contradicts our guess. We call such a testable guess about the population a ***hypothesis***.

Let's say someone has the hypothesis that the average airspeed velocity of a swallow is 22 m/s. OK, let's simulate the outcomes (the sample-means, in this case) that we would expect if that were true. We're going to play God again and now use our omnipotence to make the population-mean 22. Find the appropriate line and replace it with the code below and ***run it*** before continuing.

```
Total_number_of_swallows = 50

Speeds_Population = np.random.randn(Total_number_of_swallows)

Speeds_Population = Speeds_Population - np.mean(Speeds_Population)

Population_mean = 22

Speeds_Population = Speeds_Population + Population_mean

print(np.round(Speeds_Population, 2))

print(np.mean(Speeds_Population))
```

Run the code again and look at the list of sample-means.

Simulated sample-means:

[22.36 22.33 21.07 21.83 22.41 23.13 23.14 23.07 22.39 21.01 22.44 23.94

21.97 22.85 21.63 21.36 23.4 22.58 21.83 21.52 23.24 22.03 21.38 22.12

21.98 21.62 21.87 22.5 22.32 21.59 20.96 21.76 22.39 22.19 21.86 21.65

21.24 21.53 22.48 20.57 22.35 22.12 23.04 20.8 21.84 23.44 21.49 21.87

22. 21.44 20.99 22.58 22.71 22.41 21.99 23.11 21.95 21.12 21.89 22.77

22.33 22.03 22.14 22.64 21.92 22.13 21.77 21.72 20.68 22.29 21.2 22.27

21.82 22.53 21.92 22.28 22.93 21.94 22.08 22.69 22.25 22.6 22.4 23.

21.55 21.17 22.98 20.95 22.53 22.61 22.35 22.87 21.38 22.24 22.82 22.26

22.6 22.58 21.25 21.63]

These now cluster around 22, roughly between 21 and 23, right? Now look up the "real-life" value you found when the population-mean was set to be 11. That was 11.71, Do we ever find a value that far away from 22 in all the new sample-means? Nothing like it. That means that when we found our "real-life" value, it completely ***contradicted*** what we would expect to find ***if it were true*** that the population-mean equals 22.

In statistical terminology, we would say: we found a sample-mean that ***significantly differed*** from 22. The "significance" of an effect is the degree to which an observed effect is ***unlikely*** relative to a given hypothesis. We're seeing something surprising, something we would not expect given a belief about the population. We use this contradiction to falsify – it's evidence against the belief. Remember, we always ***falsify***, never ***verify***. At best, as sceptical scientists, we decide not to reject a hypothesis - for now.

Memorize this phrase: *Significance is about how unlikely an observation is, relative to a given hypothesis about the population.*

Memorize this phrase: *If an observation is unlikely if a hypothesis is true, then the observation contradicts the hypothesis.*

But now we need to get a more precise, objective way of measuring this contradiction between a hypothesis and an observation. It's not always going to be as obvious as in the previous example! For example, what if we want to test the belief that the average airspeed velocity of a swallow is 12? Well, adjust ***and run*** the code, again replacing the line omnipotently setting the hypothesized population speed.

```
Total_number_of_swallows = 50
Speeds_Population = np.random.randn(Total_number_of_swallows)
Speeds_Population = Speeds_Population - np.mean(Speeds_Population)
Population_mean = 12
Speeds_Population = Speeds_Population + Population_mean
print(np.round(Speeds_Population, 2))
print(np.mean(Speeds_Population))
```

My output became:

```
Simulated sample-means:
[12.2  11.51 12.99 10.85 12.99 10.87 12.31 11.1  11.54 11.79 11.86 12.82
 11.34 11.79 11.09 11.44 11.63 12.28 11.49 11.49 10.52 11.87 11.73 12.71
 10.99 11.91 12.65 11.9  12.89 11.56 11.97 11.46 12.07 11.91 12.28 12.63
 12.49 12.35 11.88 11.19 11.76 11.78 11.28 13.21 11.2  12.38 13.49 11.89
 11.46 12.46 12.47 10.91 12.02 12.02 12.34 11.61 12.31 11.86 12.73 12.54
 11.31 12.45 11.24 11.65 12.72 11.67 12.14 11.3  10.85 11.7  13.94 11.31
 11.65 11.49 11.67 12.27 10.17 11.29 11.57 11.6  11.36 12.28 12.84 12.35
 11.5  11.45 11.42 12.01 12.88 10.93 12.94 12.43 12.6  11.91 12.33 12.48
 11.47 11.53 11.73 12.07]
```

Just by eyeballing the sample-means I'm seeing a range that now easily covers the 11.71 that I observed when I did my "real-life" data collection. Would seeing that 11.71 raise a red flag of contradiction at all? How can we objectively quantify if an observation is unlikely or not?

We do this by defining lower and upper ***critical values*** of sample means, which define a range of sample means that are "suspiciously low" or "suspiciously high". Traditionally, we consider the critical values to be those such that only 5% of the sample means are either below the lower critical value or above the upper critical value. We could, however, use different percentages than 5% to define critical values that define "extremeness" more or less strictly.

Memorize this phrase: *Only a small percentage of all random samples have a sample-mean below the lower critical value or above the upper critical value.*

We can estimate the critical values using our simulations. Find and adjust the relevant code as below to store the simulated sample means in an array, instead of only printing them to screen; then we'll calculate the absolute difference of each sample-mean from the hypothesized population-mean. We want the absolute difference because we consider either abnormally low or abnormally high values

to contradict a hypothesis. We'll then sort these differences from low to high and find which one covers the top 5%. Note how I'm printing variables to screen to have an idea of what's going on.

```
# Now we're going to simulate what happens when we take a lot of different samples.
# We're going to use this to approximate the critical values.
Number_of_samples = 100
Birds_per_sample = 2
Array_of_sample_means = np.array([]);
for this_sample in range(Number_of_samples):
      My_Sample = np.random.choice(Speeds_Population, size=Birds_per_sample, replace=False)
      Array_of_sample_means = np.append(Array_of_sample_means, np.mean(My_Sample))
print('Simulated sample-means:')
print(np.round(Array_of_sample_means, 2))
Array_of_extremeness = np.abs(Array_of_sample_means - Population_mean)
Sorted_extremeness = np.sort(Array_of_extremeness)
print('Sorted extremeness of sample-means:')
print(np.round(Sorted_extremeness, 2))
Critical_value_index = int(np.floor(0.95 * len(Sorted_extremeness)))
print("Index in array containing critical extremeness: " + str(Critical_value_index))
Critical_absolute_difference = Sorted_extremeness[Critical_value_index]
Critical_value_lower = Population_mean - Critical_absolute_difference
Critical_value_upper = Population_mean + Critical_absolute_difference
print("Lower and upper critical values: " + str(np.round(Critical_value_lower, 2)) + " and " +
str(np.round(Critical_value_upper, 2)))
```

The lower and upper critical values I got were 10.17 and 13.83. Anything lower than 10.17 and anything higher than 13.83 is "extreme", i.e., is something that's less than 5% likely for me to find in a sample. My observed value of 11.71 ***wasn't extreme enough*** to fall in the critical range. So: The observed sample mean of 11.71 ***doesn't significantly differ*** from the hypothesized population mean of 12. My observed sample-mean doesn't let me reject the hypothesis that the average airspeed velocity of the swallow is 12.

Thinking in terms of critical values that tell us which ranges of extreme observations are unlikely is the basis of understanding statistical significance testing. However, what often happens using modern software seems to be a little different. We don't explicitly set our desired unlikelihood of extreme observations (usually 5%), then calculate critical values, and finally check whether the sample-mean fell in the critical range. The software will give us a so-called *p*-value (this is sometimes labelled "the significance" or "Sig.", as in SPSS). But this *p*-value actually has everything to do with critical ranges. It's telling us what the probability of extreme observations ***would have been***, ***if*** we had used the observed sample-mean (or sample-statistic in general) as the critical value. If the *p*-value is ***below*** 5%, then the critical values would be stricter, covering a smaller part of the tails of the

sample-mean distribution: an even smaller percentage of observation than 5% would be considered extreme, with the observed sample-mean as the critical value. So, if the *p*-value is ***lower than 5%***, we know the observed sample-mean would be extreme enough to be ***beyond the critical values*** that would classify 5% of sample-means as extreme. Therefore, when we run a statistical test and we get a *p*-value below .05, the effect being tested is significant.

Memorize this phrase: *The p-value tells us how unlikely the observed sample-statistic is, relative to a given hypothesis.*

Memorize this phrase: *If the p-value is below .05, there is a significant difference from the hypothesized population. This means we reject the hypothesis.*

Sometimes it's unintuitive for students that ***we have to see a small p-value*** to conclude significance, so be very careful you remember and try to understand this! E.g., if $p = .0043$, the effect is significant because .0043 is smaller than .05; but if $p = .56$, the effect is non-significant, because .56 is larger than .05.

Let's estimate the *p*-value of 11.71. We're going to use frequentism to calculate the probability of finding a sample-mean as "extreme" as 11.71, if the population-mean is 12. We can use our sorted extremeness array for this and just count. ***Add and run*** the following code:

```
Observed_sample_mean = 11.71

Observed_extremeness = np.abs(Observed_sample_mean - Population_mean)

At_least_as_extreme = np.argwhere(Array_of_extremeness >= Observed_extremeness)

Number_of_at_least_as_extreme = len(At_least_as_extreme)

p = Number_of_at_least_as_extreme / len(Array_of_extremeness)

print("The p-value of " + str(np.round(Observed_sample_mean, 2)) + " is " + str(np.round(p,
4)))

print("There were " + str(Number_of_at_least_as_extreme) + " more extreme values in the
simulated samples.")

print("There were " + str(len(Array_of_extremeness)) + " simulated values in the array in
total.")
```

I got a *p*-value of .63: it's ***non-significant***, because the *p*-value is ***too big*** (larger than .05), which agrees with the observed value ***not falling in the extreme critical range***.

This is how we can test whether a sample mean significantly differs from a hypothesized population mean! Of course, we've been using very simple simulations to approximate our probabilities, and proper statistical software uses hardcore maths and does it all far better. But conceptually it's all the same: we want to know how unlikely an observed sample-statistic would be given the hypothesis

about the population we're testing. It's very important to remember how we programmed these simulations and what we're looking at to define probabilities. The population-mean is something we control, and the probabilities of sample-means follow from that. This lets us talk about probabilities for sample-means relative to a given hypothetical population-mean; remember the way we calculated the *p*-value for an observed sample-mean above. We do not, however, have a "probability" for the population-mean – we don't have a big array of population means we're randomly selected population-means from! The population-mean is "above" probability; it's something we decide on as the God of the simulation; there is no probability even ***defined*** for the population-mean. All we do in significance testing is say: well, we can for sake of argument assume the hypothesis is true (i.e., we run our simulation with a given value for the population), and then check the probability of finding a sample-statistic as extreme as what we actually observe after data collection. We can't calculate a probability for the divine population mean in the same way we can for the sample-mean, because we don't have a big array of population means to use the argwhere() function on.

Memorize this phrase: *We can only talk about probabilities involving samples; there is no concept of probability defined for the population.*

Memorize this phrase: *The population is where the probabilities of sample outcomes comes from; the population doesn't have a probability itself.*

Of course, outside the strict confines of statistical significance testing we're certainly thinking about whether hypotheses are plausible or not. But we have to keep this strictly separate from the *p*-value, or we'll get very confused. Significance testing plays a central role in experimental research but is by no means the only thing we care about – it's one piece of evidence we can use to try to make good decisions and move towards less-wrong beliefs.

By far the most relevant kind of significance testing in experimental work is ***null hypothesis significance testing***, or NHST. This is where the hypothesis represents some kind of ***absence of an effect***. For instance, let's say that our scores represent a difference between two conditions that each participant has been exposed to, e.g., reaction times on interference versus control blocks of the Stroop task. Hypothesizing that the population-mean of the difference scores is zero formalizes and quantifies the belief that there is no true effect of Stroop interference. We then collect data and find a difference score of, say, 65 ms. Do we then conclude that there is Stroop interference? Well, no, because we know about sampling error. We need to calculate the *p*-value to be able to ***at least*** reject the null hypothesis. Note this phrasing: We don't have to believe the null hypothesis is actually true for NHST to make sense; it's often a priori implausible in real-life for there to be literally, absolutely zero difference. The reason we do NHST is to make sure that we can ***at least*** contradict the skeptic who should always live in our heads reminding us of sampling error. The less plausible the null hypothesis is in reality, the ***more*** relevant it is that we ***can't even*** reject it using our data. ***A non-significant test tells us we shouldn't be drawing conclusions about any apparent effects in our sample, no matter how much we might want to.*** Where we go from there is beyond statistics. Perhaps we try to find a more sensitive test, or we reconsider our measurements, or we

check for some source of noise we hadn’t considered; or perhaps a competing belief does lead to significant effects, in which case we’ll provisionally prefer that one, and science hopefully moves a little further towards truth.

5. Capturing more swallows

Recall the critical values we got in our simulations of a population with a mean airspeed velocity of unladen swallows - 10.17 and 13.83 for me. So anything between 10.17 and 13.83 would be non-significant. That's quite broad, and included a value I know I got from a sample with a different population with a mean of 11 m/s. Ideally, we would have detected that. Is there a way we can "tighten up" the critical values?

First, ***run*** the script again and just have a look at the sample-means you get to get a sense of how much they vary. We can quantify this variation by storing the random sample-means we get per simulated data collection, and taking the standard deviation of all those different sample-means. We already stored the sample-means in an array, so we just have to add this to the bottom of the script:

```
Standard_error = np.std(Array_of_sample_means)
print("The standard error was " + str(np.round(Standard_error, 4)))
```

Run the script again and note down this ***standard error*** – this is the ***standard deviation of sample means*** over replications of the 100 simulated experiments. For me, it was 0.96. Note how the standard error is different from the usual standard deviation of single observations within one experiment. ***The standard error quantifies sampling error***. The bigger the standard error, the more wildly the sample means fluctuate around the population mean, and the further apart will be the lower and upper critical values. And therefore, the bigger the standard deviation, the more difficult it is to get a low-enough *p*-value for an effect to be significant. If we have a very tight distribution of sample-means, then even observed values only a little way away from the population mean could become extreme.

We can reduce the standard error by capturing more birds, since in each simulated experiment we'll be basing our sample-mean on the averaging of a larger number of speeds. It will be more and more unlikely for all the sampled speeds to be very high or very low, this averaging out differences from the population mean. Let's test this claim by changing the number of captured birds in the simulation-loop and ***running*** the script:

```
Number_of_samples = 100
Birds_per_sample = 20
Array_of_sample_means = np.array([]);
for this_sample in range(Number_of_samples):
    My_Sample = np.random.choice(Speeds_Population, size=Birds_per_sample, replace=False)
    Array_of_sample_means = np.append(Array_of_sample_means, np.mean(My_Sample))
```

This reduced the standard error to 0.16 and tightened the critical values to 11.65 and 12.35. My 11.71 still wouldn't reach significance but you can see it's getting closer; and its *p*-value was indeed now lower, .1 instead of .63.

There are different, better ways of simulating data of course. We're creating an actual, relatively small population that lives in our computer memory. Usually the population of interest would be an infinite, mathematically defined probability distribution, and we'd use a random-value function to generate samples from it. To draw a sample of 20 values from a uniform distribution from -1 to +1, for instance, we'd use:

```
this_sample = np.random.uniform(-1,1,20)
```

To draw a sample of 20 values from a standard normal distribution, we'd use:

```
this_sample = np.random.randn(20)
```

An important conceptual point is that the *p*-value of a certain value of a sample-statistic depended on a ***parameter***; namely, the number of birds captured per experiment, i.e., the sample size. That makes sense: the more birds we capture, the smaller the standard error, and the lower the *p*-value. Now, remember those degrees of freedom, *df*? Those are simply the parameters of a statistical test that determine what the *p*-value of a given observed value is. Just like in our example, we ***need to know the parameter*** of Birds_per_sample to be able to say what a given sample-mean's *p*-value is – the *p*-value was .63 when Birds_per-sample was 2, but .1 when Birds_per_sample was 20. Why are they called "degrees of freedom"? The relevant parameters for the hardcore maths underlying to tests have to do with how many relevant values depend on random selection and thus "freely vary" over different samples. Unless you go into the mathematic of statistics, *df* are just a kind of nasty exposure of how the sausage is made, but traditionally you have to be able to calculate them per test and report them. They do allow for some quality and sanity checks, but, in my personal opinion, it would be a lot nicer if we reported the basic parameters like sample size and number of groups rather than the quasi-technical *df* (which are completely determined by such understandable parameters).

A further essential point is that small samples are deadly (Button et al., 2013). Nothing you can do can save you from the realities of sampling error and high standard error you get with small samples. In particular, you can't compensate by some vague notion of replacing "breadth" by "depth" – if you include many variables instead of just one, your problem hasn't gotten better: each of the variables will be noisy. If you try to compensate by invoking a priori beliefs, at some point you might as well not have bothered gathering the data.

Finally: if you want to get the *p*-value of a sample with a proper statistical function, you can use the scipy package. At the top of the file, add:

import numpy as np

from scipy import stats

Where you created a single sample, add:

```
# Now we're going to draw a random sample of two swallows.
My_Sample = np.random.choice(Speeds_Population, size=2, replace=False)
print('The captured swallows had speeds:')
print(np.round(My_Sample, 2))
print('The average speed is this sample was:')
print(np.round(np.mean(My_Sample), 2))
test_results = stats.ttest_1samp(My_Sample, Population_mean)
print(test_results)
```

This will give you output including the *p*-value:

Ttest_1sampResult(statistic=23.76999129107635, pvalue=0.02676671540692633)

6. Other statistical tests

In the preceding sections, I've tried to illustrate the statistical concepts of inference from sample to population, sampling error, null hypothesis significance testing and *p*-values within the context of tests of sample-means. This is what a one-sample t-test or paired-sample t-test does, using proper software. The same concepts generalize to all the usual tests, except the null hypothesis – the thing our inner skeptic demands we ***at least*** need to reject to even ***think*** about looking at patterns in our sample – will mean something different. I'll briefly go over the very basic tests, but as an exercise for the reader, if there's one you really want to understand, do the same simulation trick as I went through above.

Between-group t-test

The null hypothesis is that ***two groups*** have the same population means. A significant effect ($p < .05$) means that the size of the difference between the groups is so extreme it contradicts the null hypothesis. Therefore: you can reject the null hypothesis and move forward to interpret the effect you're seeing as if it says something true. To simulate this, we need to define two population means, as follows using multi-dimensional arrays where the first dimension is the group:

```
N_per_group = 50

Population_means = np.array([100, 110])

Scores_per_group = np.random.randn(2, N_per_group)

for index_group in range(2):

    Scores_per_group[index_group] = Scores_per_group[index_group] -
np.mean(Scores_per_group[index_group])

    Scores_per_group[index_group] = Scores_per_group[index_group] +
Population_means[index_group]

print(np.round(Scores_per_group, 2))

for group_array in Scores_per_group:

    print(group_array)
```

When running the simulation loop, we would take random samples from each group, and calculate the sample-difference between the two sample-means. This would give us the hypothesis-generated distribution of sample-differences to use for significance testing of an observed sample-difference.

```
Number_of_samples = 100

sample_size = 10

Array_of_sample_differences = np.array([]);

for this_sample in range(Number_of_samples):

    sample_means = np.array([])

    for group_index in range(2):

        sample_from_group = np.random.choice(Scores_per_group[group_index], size=sample_size,
replace=False)

        sample_means = np.append(sample_means, np.mean(sample_from_group))

    print(sample_means)

    sample_difference = sample_means[1] - sample_means[0]

    Array_of_sample_differences = np.append(Array_of_sample_differences, sample_difference)

print('Simulated sample-differences:')

print(np.round(Array_of_sample_differences, 2))
```

The official function to test group differences from scipy is scipy.stats.ttest_ind, which takes the two arrays containing the samples from each population and does an independent-samples t-test.

Analysis of Variance (ANOVA)

The null hypothesis is that ***two or more groups*** all have the same population means. A significant effect can therefore mean lots and lots of things: Maybe one group differs from all the groups, maybe groups 1 and 2 differ from groups 3, 4, and 5, and so on. You'll need to run so-called post-hoc tests to figure out what pattern caused the significant ANOVA test. What approach makes sense will depend on your context. In experimental work, ***"protected"*** post-hoc tests seem most appropriate – this is where we just test whatever we like, e.g., pairwise t-test, and don't worry about multiple-testing correction. Our aim is ***descriptive*** – we know there's a significant effect already from the ANOVA, we're just using tests to help characterize the pattern. However, if you're really interested in formally testing specific group differences, this of course isn't appropriate and you'll need to use a multiple-testing correction method (see, e.g., NumNomS, https://github.com/thomasgladwin/NumNomS, for my improved version of Bonferroni correction).

The simulation is the same as for two groups, just with three or more dimensions to the array and associated population means:

```
N_per_group = 50

N_groups = 3

Population_means = np.array([100, 100, 100])

Scores_per_group = np.random.randn(N_groups, N_per_group)

for index_group in range(N_groups):

    Scores_per_group[index_group] = Scores_per_group[index_group] -
np.mean(Scores_per_group[index_group])
```

```
    Scores_per_group[index_group] = Scores_per_group[index_group] +
Population_means[index_group]

print(np.round(Scores_per_group, 2))

for group_array in Scores_per_group:

    print(group_array)
```

The statistic to measure in the simulation loop is the variance of sample means (since we have more than two group, we can't use a simple difference score), which gets bigger the less equal they are. This therefore measures the "effect" in ANOVA, i.e., the deviation from the null hypothesis.

```
Number_of_samples = 100

sample_size = 10

Array_of_sample_variances = np.array([]);

for this_sample in range(Number_of_samples):

    sample_means = np.array([])

    for group_index in range(N_groups):

        sample_from_group = np.random.choice(Scores_per_group[group_index], size=sample_size,
replace=False)

        sample_means = np.append(sample_means, np.mean(sample_from_group))

    print(np.round(sample_means, 4))

    sample_variance = np.var(sample_means)

    print(np.round(sample_variance, 4))

    Array_of_sample_variances = np.append(Array_of_sample_variances, sample_variance)

print('Simulated sample-variances:')

print(np.round(Array_of_sample_variances, 2))
```

And again, we have a hypothesis-generated array containing simulated sample-variances; and we can use this for significance testing. If we take of sample of the three groups, we can say whether the sample-variance is extreme relative to this hypothesis. (In this case, it was the null hypothesis: the three population means were all 100.)

For an official scipy function, see scipy.stats.f_oneway. Note that the mathematical way to assess significance is based on the distribution of the *F*-statistic. The subtlety of *F* is that it's a ratio of mean squares – hang on, those aren't actually, to be precise, variances! What's going on there? Well, those mean squares are two different estimates of the population variance, one based on the variance of group means and the other based on the variance of individual scores within groups. Under the null, these should give the same estimate of the population variance. If the null is false, the group means will start varying more than they "should" and give a bigger estimate than the within-group variance of individual scores. Hence, "Analysis of Variance" should really be called "Analysis of Mean- versus Individual-based Population Variance Estimates" or AMIPVE.

Correlation

The null hypothesis for correlation is that there is no linear relationship between two approximately continuous variables X and Y. A (two-sided) significant effect means there's either a positive/negative relationship. You're probably fine just visualizing the usual kind of scatterplot here, but it's formally quantified as: over all data points in the population, each point being a pair of values on the variable X and Y, values that are higher/lower than the mean of X tend to be *proportionally* higher/lower than the mean of Y. (For a nonlinear kind of correlation, just for fun if you're interested, see nncorr on https://github.com/thomasgladwin/nncorr.) Let;'s explore what that "tend to be" means. If we imagine a set of observations on a vertical slice of an idealized scatterplot, all of them will have an x-value of mean(x) + d_x, for whatever deviation from the mean d_x. Their mean y-value will be mean(y) + d_y, for some d_y. In Python, we'd now save the ratio d_y/d_x for that slice. Now pick a different vertical slice – now we have a different d_x, and we'd find a different d_y, for that slice, and a different ratio d_y/d_x. If we repeat that for all the possible vertical slices, and take the mean of the ratio's, that tells us how y tends to be proportionally related to x. We can't do that perfectly, but maths can, and lets us define the correlation as the covariance between X and Y divided by the product of the standard deviations of X and Y.

Let's see if we can simulate correlated variables. For the simple case of two standard normal variables (i.e., mean 0 and standard deviation 1), X and Y, a correlation between them can be created by splitting Y into two uncorrelated parts Y1 and Y2 (so their variances can be summed), one of which (say, Y2) is uncorrelated with X. So, Var(X) = Var(Y1) + Var(Y2). Var(X) = 1 since it was standard normal. If Var(Y1) = r^2 and Var(Y2) = (1 – r), then the explained variance of Var(X) has to be r^2, since only Y1 predicts variance in X. Therefore, the correlation between X and Y must be r. This lets us construct a variable Y relative to a random array X and a specified correlation r. The function uses linear regression, discussed further below, to get the perfectly-correlated-with-X Y1 (the predictions of Y) and uncorrelated-with-X Y2 (the residuals, or error, between observed and predicted values of Y).

```
import numpy as np
from scipy import stats
def create_correlated_variable(X, r):
    N = len(X)
    Y_init = np.random.randn(N)
    slope, intercept, r_value, p_value, std_err = stats.linregress(X, Y_init)
    Y_pred = intercept + slope * X
    Y_resid = Y_init - Y_pred
    Y_pred = stats.zscore(Y_pred)
    Y_resid = stats.zscore(Y_resid)
    Y_correlated = r * Y_pred + np.sqrt(1 - r**2) * Y_resid
    return Y_correlated
N = 50
```

```
r = 0.35
X = np.random.randn(N)
Y = create_correlated_variable(X, r)
test_r, p = stats.pearsonr(X, Y)
print("We want the correlation to be " + str(r))
print("We made the correlation " + str(test_r))
```

The scipy function for correlation is scipy.stats.pearsonr.

Regression

When you visualize a correlation scatterplot and a line drawn through it to represent the linear trend, that line is the ***regression line***. That's the line, a function y_predicted = b*x + c, that best minimizes the mean over all points of the squared difference between the observed (x, y_observed) dots and the corresponding points on the line (x, y_predicted). This line is created by simple linear regression, where we're optimizing how well we can predict the y-values from the x-values using just that linear model. If you look at the *p*-values for a correlation between two variables, you'll see it's identical to the *p*-value of the regression: it's really the same null hypothesis being tested in the same way.

When we predict y from a single predictor x this is called "simple regression", which tells us nothing more about their relationship than a correlation. In general, we do multiple regression rather than simple regression, where "multiple" means we have more than one predictor "x", but a whole set of predictor variables x_1, x_2, …, x_n; we still only have one single predicted value, y. The essential thing to remember is that there are two kinds of tests in regression: First, we have the *F*-test that tests whether ***the whole model*** explains more sample-variance than expected under the null hypothesis that the model explains nothing in the population. In other words, the null hypothesis is that the true model has coefficients of zero for all predictors. In finite samples of observed data, there'll always be some spurious, sample-specific relationship that regression will exploit to explain some non-zero amount variance; the *F*-test tells you whether a sufficiently extreme amount of sample-variance is explained. The idea of "explained variance" comes from being able to write raw scores of the dependent variable as the sum of two ***uncorrelated*** components: predicted values and the error. Because in regression these components are uncorrelated, their variances must sum to the variance of the original scores, and we can just calculate the variance of the predicted values and see what its proportion is of the total original variance – this has to be (barring computational foibles perhaps) some proportion from 0 to 1. Second, we have the set of *t*-tests that test whether the coefficient of ***each specific predictor separately*** differs significantly from zero.

Let's have a play around with regression. We start off with the basic linear model connecting the variables y (the dependent variable) and x (the predictor):

$$y = b*x + c$$

So, in Python, x and y would be arrays of the same length, and the linear model tells us what a given element y[i] in y is, if we know the corresponding element in x, x[i]. We have two *coefficients* that determine this model: b and c. These are single, fixed numbers, that connect the pairs of values in the variables x and y. The coefficient, or parameter, c is the offset – what value does y have when x is zero? I.e., if we draw the line, where does it "start" on the graph where it cuts the y-axis? The coefficient b is the one we're usually really interested in: how much does y change if x increases by 1?

So let's make this in code, using b = 2 and c = 5. I'm using the Pandas ("PANel Data AnalysiS") library here to easily plot a matrix of values.

```
import numpy as np
from scipy import stats
import pandas as pd
b = 2
c = 5
x = np.array([-1, 0, 1, 2, 10])
y = x * b + c
DataFrame = pd.DataFrame({'x':x, 'y':y})
print(DataFrame)
```

The output should be:

```
    x   y
0  -1   3
1   0   5
2   1   7
3   2   9
4  10  25
```

Make sure you understand the link between the x and y column: when x is 0, y = 5 – yes, that's the offset. If x goes up to 1, y goes up to 5 – yep, 2 extra. And so on.

Now, say that we have two variables like x and y, and we know there's a linear relationship between them, but we don't know the coefficients. Let's see if regression can tell us what those generating coefficients were. Add this to the previous code creating x and y. I'll use the scipy function for linear regression, scipy.stats.linregress.

```
slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)
print('c = ' + str(intercept))
print('b = ' + str(slope))
```

This should return the 5 and the 2, possibly with some annoying tiny deviations due to the technical implementation. Try a few different coefficients to convince yourself we can estimate the coefficients back from the generated data.

With real data, y won't be a perfect linear function of x. The actual relationship will have an error component, e, in addition to the linear model part kept in parentheses:

$$y = (b*x + c) + e$$

In linear regression, the coefficients are chosen to minimize the variance of e, and e is defined to have a mean of zero. That's enough to nail down the coefficients.

Where it gets very, very, very tricky is when we expand the formula to have multiple predictors x_1, x_2, etc:

$$y = b_1*x_1 + b_2*x_2 + b_3*x_3 + c + e$$

Each predictor now has its own coefficient to estimate. Each coefficient has its t-test against zero; there's still one overall F-test for the whole model. This is all well and good if all the predictor are mutually independent – "independent" being an even stricter version of "uncorrelated", where every possible transformation of the variables still results in a correlation of zero, and we can explain zero variance of any predictor when using the others. In that very special and totally unrealistic case, we can recover coefficients as expected. But note that we're really making an effort to get those independent predictors:

```
import numpy as np
from scipy import stats
import pandas as pd
b1 = 2
b2 = 1
b3 = 4
c = 5
N = 1000

# We pick the values of variable x1 at random
x1 = np.random.randn(N)
# Now we construct x2 to be independent from x1
# We replace it with its own residuals from its linear prediction.
```

```
x2 = np.random.randn(N)

slope, intercept, r_value, p_value, std_err = stats.linregress(x1, x2)

x2 = x2 - (intercept + slope * x1)

# And we construct x3 to be independent from x1 and x2

# This requires a more advanced regression function than the simple linregress. I'll use one from NumPy.

# Note I'm manually adding a column of 1's - the coefficient for these will be the offset.

x3 = np.random.randn(N)

X = np.vstack([x1, x2, np.ones(len(x1))]).T

R = np.linalg.lstsq(X,x3)

coeffs = R[0]

x3_pred = coeffs[0] * x1 + coeffs[1] * x2 + coeffs[2];

x3 = x3 - x3_pred


# Test whether x1, x2 and x3 are indeed uncorrelated

# Note that a weird-looking number like 3.23222e-17 is pretty much zero - the "e-xx" means 10 to the power of -xx; so for instance, 5e-6 means 0.000005.

stats.pearsonr(x1, x2)[0]

stats.pearsonr(x1, x3)[0]

stats.pearsonr(x2, x3)[0]


# Now we can make y be a linear function of x1, x2 and x3

y = b1*x1 + b2*x2 + b3*x3 + c


# Now we do the multiple linear regression again, now with y as the dependent variable

X = np.vstack([x1, x2, x3, np.ones(len(x1))]).T

R = np.linalg.lstsq(X,y)


print('b1 = ' + str(R[0][0]))

print('b2 = ' + str(R[0][1]))

print('b3 = ' + str(R[0][2]))

print('c = ' + str(R[0][3]))


# Compare these values with the simple regression slopes

slope_b1, intercept, r_value, p_value, std_err = stats.linregress(x1, y)

slope_b2, intercept, r_value, p_value, std_err = stats.linregress(x2, y)

slope_b3, intercept, r_value, p_value, std_err = stats.linregress(x3, y)


print('Simple: b1 = ' + str(slope_b1))

print('Simple: b2 = ' + str(slope_b2))
```

```
print('Simple: b3 = ' + str(slope_b3))
```

So we get the right coefficients, again, maybe with tiny deviations, and there's no difference with separate simple regression. Try it for some different coefficients.

But imagine what would happen if x_1 and x_2 were highly correlated, while each, separately, still having the same linear relationship with y. Any such lack of independence is called ***multicollinearity***. We'll create this situation below. Look at what happens in the multiple regression.

```
import numpy as np

from scipy import stats

import pandas as pd

b1 = 2

b2 = 1

b3 = 4

c = 5

N = 1000


# Now x1 and x2 are going to be correlated, since x2 is going to be defined as its initial
random values plus x1. The variable x3 isn't made *exactly* independent by replacing it with
residuals but since we have a large sample it'll be pretty well uncorrelated just by being
random.

x1 = np.random.randn(N)

x2 = np.random.randn(N)

x2 = x2 + x1

x3 = np.random.randn(N)


# Check out the high correlation between x1 and x2

stats.pearsonr(x1, x2)[0]

stats.pearsonr(x1, x3)[0]

stats.pearsonr(x2, x3)[0]


# Now we make y be a linear function of x1, x2 and x3

y = b1*x1 + b2*x2 + b3*x3 + c


# Now we do the multiple linear regression

X = np.vstack([x1, x2, x3, np.ones(len(x1))]).T

R = np.linalg.lstsq(X,y)


print('Multi: b1 = ' + str(R[0][0]))
```

```
print('Multi: b2 = ' + str(R[0][1]))
print('Multi: b3 = ' + str(R[0][2]))
print('Multi: c = ' + str(R[0][3]))

# Compare these values with the simple regression slopes
slope_b1, intercept, r_value, p_value, std_err = stats.linregress(x1, y)
slope_b2, intercept, r_value, p_value, std_err = stats.linregress(x2, y)
slope_b3, intercept, r_value, p_value, std_err = stats.linregress(x3, y)

print('Simple: b1 = ' + str(slope_b1))
print('Simple: b2 = ' + str(slope_b2))
print('Simple: b3 = ' + str(slope_b3))
```

While the simple regressions give up back something still very close to the coefficients we used to create y, the coefficients for x1 and x2 in the multiple regression can be completely different. It's mathematically impossible for the regression calculations to separate out the contributions of x1 and x2: they contain the same shared variance. All the regression can do is give us the coefficients that, working together for the all the predictors at once, minimize the overall error. Nothing whatsoever guarantees that those coefficients will be interpretable in terms of the simple pairwise relationship between one particular predictor and the dependent variable. There's often a huge problem with looking at the individual *t*-tests to draw conclusions because of multicollinearity – the specific individual predictors will interfere with each other, unless they're stochastically independent. This is just what regression does: it will by definition find the best linear combination, and that means juggling the whole set of coefficients to suit that aim and that aim alone.

The main way I use regression is therefore ***hierarchical regression***: I start with a baseline model of predictors unrelated to my hypothesis and use an *F*-test to see whether an expanded model with additional hypothesis-related predictors explains a significant amount of ***additional*** variance, or "unique" variance. This lets me test whether the additional *set* of predictors explains a significant amount of additional variance. Only if a set explains significant additional variance in the *F*-test, I'll look at the coefficients in the model in combination with the simple pairwise correlation and try to make a careful interpretation based on that. People have made up rules that say interpreting individual predictors within a broader model is OK if certain metrics, like the Variance Inflation Factor, are within certain ranges, but these rules seem to have generalized validity if you check out the statistical literature – it's safer to assume that if you have any multicollinearity, you're *always* on thin ice when naively interpreting individual coefficients. In some cases, it may make sense to create a set of independent predictors via principal component analysis, but the problem then is likely to be what each component means.

Finally, a brief mention of interaction effects. This is when a slope-coefficient is itself a linear function of a predictor:

$y = b1 * x1 + c1$

$b1 = b2 * x2 + c2$

So, putting the equations together,

$y = (b2 * x2 + c2) * x1 + c1$

$y = b2 * (x2 * x1) + c2 * x1 + c1$

So we have a multiple regression situation, with a special predictor that consists of the product of x1 and x2. If the coefficient for that predictor (or "interaction term") is significant, then that means that the value of b1 (and hence the relationship between x1 and y) significantly varies with the value of x2.

7. Conclusions

In conclusion, I hope this text has provided some helpful concepts, maybe some useful snippets of Python, and an introduction to simulations as a general trick for understanding things. You may even have thought of ways to make better simulations to actually do significance testing for non-standard analyses. This trick has let me develop analyses for Genome-wide Association Studies that were familywise error-corrected but didn't require obscenely low *p*-values per SNP; tests of time series of heart rate and body sway data; whole-brain tests of activation blobs in fMRI; and tests of white matter tract abnormalities. Whatever the data pattern: If you can simulate it, you can permute it, and then you can significance test it.

In terms of philosophy of science, scientists doing (good) science embody a great algorithmic machine that implements the hypothetico-deductive (or simply "scientific") method of hypothesis testing, an empirical loop that tends to detect and reject errors and thereby implicitly moves our beliefs towards truth. To avoid confusion, there are two contexts of hypothesis testing we need to clearly distinguish. There's the "technical" deduction of what is likely under the null hypothesis, which we have to do that to silence the automatic inner skeptic that pops up anytime we look at what seems to be an effect. But we also have the "positive" hypothesis or the "hypothesis of interest" that we really want to test our belief about (technically and unhelpfully called the "alternative hypothesis"). This opens up a whole different story from merely NHST. The machinery of a merely statistical test works only in one direction: we can reject a hypothesis if the data significantly contradict it. But we can't twist that around and say: we can accept a hypothesis as true if the data do not significantly contradict it. That would be easy! We would just run very underpowered studies with huge standard errors that couldn't contradict anything. What we have to undertake to test positive hypotheses is a much higher-level, integrated strategy. We never believe a hypothesis is absolutely true, since they're just models and we're not dumb enough to think we've figured out the whole universe; but we can believe they're a step towards reality in all its infinite-dimension-ness. Testing our hypothesis of interest involves a more complex level of argumentation

that combines many sources of information and influence. To support our hypothesis, we have to demonstrate we can systematically produce interesting and statistically significant results that conceptually agree with our proposed causal model better than with competing models. ***To fail to produce evidence is to falsify our claim we can show evidence for our hypothesis***, which happens over time and over multiple studies; it involves the rise of alternative models and new methods; and it involves judgments and goals – a claim can become non-interesting rather than failing to accurately predict data. Thus, the statistical method of null hypothesis significance testing lives within the broader scientific context of hypothesis falsification, but in an indirect and constrained way.

Memorize this phrase: *Lack of evidence is sufficient to provisionally falsify a hypothesis of interest*

Memorize this phrase: *Evidence is found by demonstrating contradiction with the null hypothesis*

Memorize this phrase: *If you can't even reject the null hypothesis, then the Great Science Machine will rightfully reject your ideas*

You might have noticed that I'm working from the perspective of experiment research, where we expect to do many different studies on a particular topic and our aim is to test ideas. For this kind of research, NHST is perfectly valid: we're likely to try a lot of things that don't work and we want a feedback mechanism telling us whether we're being pointed in the right direction or just getting fooled by fake patterns. ***The reason we use p-values is to stop ourselves following will-o'-the-wisps in the data*****.** Other kinds of research and researchers might well prefer different kinds of statistics, for instance, if instead of smaller studies that ideally should be replicable you have one large, possibly unique dataset; or when instead of testing which ideas are best, you really just want to say something about the actual population values. These are very different aims, related to "***epidemiological" versus "experimental"*** research perhaps. I suspect this is where much of the sometimes really irrationally negative vilification of NHST comes from: Of course NHST isn't the right tool to use if your whole aim is to estimate the actual numeric population values and quantify uncertainty about them. Sure, go for confidence intervals and Bayesian statistics if they suit your purposes better. But don't be a fanatic or a Reviewer #2 about it towards experimentalists, and don't confuse the issue with strawmen. Students, and plenty of academics, are confused enough about statistics, and the warfare with its associated rhetoric between methodological approaches and preferences often seems more counterproductive than clarifying. All the serious statistical problems related to statistical testing, such as in the replication crisis in psychology, obviously involved the more widespread NHST, but these problems could easily translate to and infect other approaches to statistics. Any metric dependent on the sample is vulnerable to changing the sample inappropriately and all the dishonest methodological and pre-processing tricks to get a significant *p*-value could be used to get a more preferred result in Bayesian analyses. I would argue that, in the debate on how psychologists should analyze their data, conceptual and mathematical simplicity of statistical approaches should be seen as a desirable dimension; obviously in the context of the necessity that results are valid. First, there are limited resources in science and spending them on dealing with unnecessary statistical barriers is unethical; second, overly complicated methods

requiring highly specialist expertise to be used reduce the level of competence and responsibility of researchers and hence could well introduce errors. I recall, for example, a “fancy” method that almost everyone used but almost nobody could understand, and that turned out to be wrong only after years and years. For many purposes, NHST is simple and valid and lets experimental researchers get on with all the other elements of their work. In closing, I’d therefore raise the concern that well-meaning but heavy-handed imposition of new statistical rules, e.g., by journals, may be inappropriate and counterproductive. Keep it Simple, Statistically.

8. References

Button, K. S., Ioannidis, J. P. A., Mokrysz, C., Nosek, B. A., Flint, J., Robinson, E. S. J., & Munafò, M. R. (2013). Power failure: Why small sample size undermines the reliability of neuroscience. *Nature Reviews. Neuroscience*, *14*(5), 365–76. https://doi.org/10.1038/nrn3475

Deutsch, D. (1997). *The Fabric of Reality: The Science of Parallel Universes—And Its Implications*. Penguin Books. https://www.amazon.com/Fabric-Reality-Parallel-Universes-Implications/dp/014027541X

Popper, K. (1934). *The Logic of Scientific Discovery*. Routledge.

Wigner, E. P. (1960). The unreasonable effectiveness of mathematics in the natural sciences. Richard courant lecture in mathematical sciences delivered at New York University, May 11, 1959. *Communications on Pure and Applied Mathematics*, *13*(1), 1–14. https://doi.org/10.1002/cpa.3160130102