Did Dream Cheat? A Bayesian Analysis of a Bernoulli Process.

hjhornbeck

Feb 05, 2021

CONTENTS

I know of two analyses of Dream's recent speedrunning performance, one from the Minecraft Speedrunning Team, the other from Photoexitation. While both make some good points, both also have serious flaws. This document is my attempt to do a better analysis.

In *A Critique of the Reports*, I point out a number of issues with those two analyses, the most notable of which are the Minecraft Speedrunning Team's *reliance on p-values* and Photoexcitation's failure to mention that *Bayesian methods are inherently less likely to conclude Dream cheated*. I walk through the logic and mathematics behind my approach in *Methodology*, which relies heavily on Bayesian statistics and conjugate priors. I implement said math in two Python routines and in *Simulation Results* I validate them through simulations, with the goal of allowing anyone to easily run their own analysis.

In *Real-World Results*, I find the probability of Dream cheating to be much lower than the Minecraft Speedruning Team's report, at least five orders of magnitude lower in fact. I also find his performance to be more consistent with simulated players who modified their version of Minecraft, and inconsistent with simulated players with unmodified Minecraft. Dream's performance is also inconsistent with comparable real-world Minecraft speedrunners. *I do not make a definitive conclusion*, though, in deference to the Minecraft speedrunning community. In *Future Work*, I suggest a few ways to extend and improve this analysis.

In *Cheating Techniques* I also point out an apparent oversight in the speedrunning community which could allow for different styles of cheating that could easily be missed. I outline *how to detect these styles*.

CHAPTER

ONE

INTRODUCTION

1.1 Motivation

I first became aware of the accusations against Dream via a blog I follow. As a casual Minecraft player who has watched some of Dream's videos, I was drawn to read the Minecraft Speedrunning Team's report. It is very well written, in my opinion, and the team has put a lot of time and effort into creating a thorough investigation. Nonetheless, it made several flawed assumptions that create fundamental problems with the analysis.

That same blog post pointed me towards a rebuttal report, which had been declared "hilariously bad" and riddled with "absurd errors." My primary critiques of the MST's report were not mentioned, oddly, which drew me to read the rebuttal. Photoexcitation's report also made some faulty assumptions, and again the most important ones were missed by the critiques. While not perfect by any means, the second report was better than I was led to believe.

I asked myself if I could create a better report. After some thought, I realized the situation was quite similar to others I've analyzed in the past, and so I started work on what you are currently reading.

1.2 Declaration of Bias

I am a graduate student in computer science, specializing in computer graphics. I have not taken a university-level statistics course, though I have taken two courses on cryptography and information theory, and those subjects rely heavily on probability. Neither taught me anything new about probability; my interest in algorithms and philosophy have led me to casually study the statistics underlying science. I've found that self-study to be of immense use, as all my research has relied on statistical analysis at some level. At best, I can be considered an amateur statistician.

My plan to compensate for my lack of authority is to be more detailed and open. I will walk through all the math I employ, and explain my analysis and logic as I go. Not only is all the code I used available via GitHub under the GNU Public License 3.0, it is present in this document as well. You can edit and replay both the simulations and analysis via a webbrowser, usually with a click of a button. Pages that support it will have a rocket icon along the page header.

I have not been paid by Dream to write this report, directly or indirectly. I have not been contacted by him or any third party, let alone asked to write this report. I have enjoyed watching some of his videos, and I believe he is a very talented Minecraft player, but I am indifferent to whether or not his speed runs are considered valid.

By the same token, I have no relationship with the Minecraft speedrunning community. None of them have contacted or paid me to write this report, directly or via a third party. I have no plans on becoming a Minecraft speed runner, I'm content to merely build things and do some casual streaming. They have no more influence on me than Dream.

In short, my motivations for writing this report can be boiled down to two points:

1. I hate p-values.
2. I enjoy working with conjugate priors.

CHAPTER

TWO

A CRITIQUE OF THE REPORTS

2.1 Minecraft Speedrunning Team

The MST report is quite good. The authors go into extensive detail on how Blaze rod drops and Pigmen barters for Ender pearls work, to the point of analyzing how random numbers are generated within Minecraft. They gather data on the performance of other runners, which is vital to help rule out flaws in your analysis: if the analysis shows *all* of the top players are cheating, then either cheating is rampant or something is wrong with your analysis. The report goes into quite a bit of detail on the methods used, and shares code to help with reproducibility. I believe it is a good attempt to "present [an] unbiased, rigorous statistical analysis of the data" (pg. 4).

2.1.1 Inadequate Comparisons to Other Speedrunners

The MST report links to raw data for five top-tier speedrunners, only one of which is Dream. (pg. 24). Rather than carry out their analysis on all five speed runners, though, only Dream's statistics are considered in depth. Figures 1 and 2, on pages 5 and 6, are the only attempt to compare Dream's performance to that of other top speed runners, and even then only raw data is displayed rather than test statistics. The authors of this report had the means and ability to calculate their test statistic for all five runners, and yet they either did not or did not show it. This leaves them wide open to charges of bias.

Running other speedrunner's performance through their metrics is critical to ensure they are working properly. This is one reason why scientific studies usually have control groups, even though it requires twice the effort of not running a control.

2.1.2 P-values are Not What You Think

The biggest flaw with the report is the reliance on p-values to generate its conclusions. I have summarized many critiques of that statistic elsewhere, but two points are worth covering in detail here.

This is the definition of the p-value:[Goo99]

> If we assume the null hypothesis is true, and draw an infinite amount of data like what has been observed, what proportion of time would the calculated test metric be equal to or more extreme than the test metric predicted by the null hypothesis?

Modus Tollens

- If A is true, B is true.
- B is false.
- Ergo, A is false.

On its surface, it's a very strange metric. Why are we basing our decision on data we have not and will never observe? What counts as "extreme"? The core logic relies on both modus tollens and the Central Limit Theorem, which are usually correct but not always so.[Coh94][Van14]

Central Limit Theorem

The means of sufficiently large samples taken from a larger population will follow a Gaussian distribution centred at the population mean.

P-values are not a probability of the null hypothesis being incorrect and cannot be treated as such, yet the MST report declares their final p-value to be an "upper bound on the chance" of Dream getting that lucky (pg. 22). The text which popularized the p-value encourages this mistake.

> The term Goodness of Fit has caused some to fall into the fallacy of believing that the higher the value of P the more satisfactorily is the hypothesis verified. Values over .999 have sometimes been reported which, if the hypothesis were true, would only occur once in a thousand trials. Generally such cases are demonstrably due to the use of inaccurate formula, but occasionally small values of χ^2 beyond the expected range do occur, as in Ex. 4 with the colony numbers obtained in the plating method of bacterial counting. In these cases the hypothesis considered is as definitely disproved as if P had been .001. (pg. 96)[F+34]

Sir Ronald Fisher himself sometimes presents the p-value as the odds of the null hypothesis being true or false. Textbook authors carried the error forward.

> What Guilford teaches as the logic of hypothesis testing is Fisher's null hypothesis testing, deeply colored by "Bayesian" terms: Null hypothesis testing is about the probability that the null hypothesis is true. "If the result comes out one way, the hypothesis is probably correct, if it comes out another way, the hypothesis is probably wrong" (p. 156). Null hypothesis testing is said to give degrees of doubt such as "probable" or "very likely" a "more exact meaning" (p. 156). Its logic is explained via headings such as "Probability of hypotheses estimated from the normal curve" (p. 160). Guilford's logic is not consistently Fisherian, nor does it consistently use "Bayesian" language of probabilities of hypotheses. It wavers back and forth and beyond.[Gig93]

Even experts can misinterpret it.

> In a recent survey of medical residents published in JAMA, 88% expressed fair to complete confidence in interpreting P values, yet only 62% of these could answer an elementary P-value interpretation question correctly. However, it is not just those statistics that testify to the difficulty in interpreting P values. In an exquisite irony, none of the answers offered for the P-value question was correct . . . [Goo08]

2.1.3 P-values Exaggerate the Evidence

Multiple studies have looked into the predictive power of p-values by simulating experiments.[HL08][Col14][HCEVD15] These should be taken with a grain of salt, as many of them argue p-values exaggerate the evidence relative to Bayesian methods. If you believe hypotheses can only be true or false, and cannot be assigned probability values, then you believe Bayesian methods are invalid and thus cannot be used as a reference. Not all of these simulation studies make the same mistake, though.

> In this article, I investigate p-values in relation to replication. My conclusion is that, if you repeat an experiment, you are likely to obtain a p-value quite different from the p in your original experiment. The p-value is actually a very unreliable measure, and it varies dramatically over replications, even with large sample sizes. Therefore, any p-value gives only very vague information about what is likely to happen on replication, and any single p-value could easily have been quite different, simply because of sampling variability.[Cum08]

The evidence is strong enough that some researchers have proposed lowering the default p-value cut-off of 0.05.[Joh13] The theory is that rather than asking scientists to abandon the use of p-values, it is simpler to lower the threshold for significance and thus increase the amount of evidence required to reach that bar. Some fields of science have already made that transition, for instance particle physics and genomics research often require $p < 0.001$ or even $p < 0.0000003$.[Lyo13][ST03] If p-values exaggerate the evidence against the null hypothesis, and the null hypothesis is that Dream's drop rate was unmodified, then by using p-values the MST report is exaggerating the evidence against Dream. This alone makes another report worthwhile.

I have other complaints about the MST report, but there's significant overlap between them and what's in the Photoexitation report.

2.2 Photoexcitation

The PE report offers a few good critiques. It argues that since Minecraft speedrunners almost always stop after they've earned k items, instead of stopping after n attempts to earn those items, the resulting k and n do not follow the binomial distribution. The MST report does not consider this argument in the context of items picked up, though it does consider it in the context of when a speedrunner stops playing.

The PE report's decision to use Bayesian statistics was correct, as in my opinion it copes better with small sample sizes than frequentist methods such as the p-value. It also used simulations to examine the relevant mechanics of Minecraft, a wise choice that makes it easier to test different possible outcomes in a controlled environment.

2.2.1 Overuse of the Negative Binomial

Minecraft Versions

Minecraft comes in several variations, and some variations make it easy to run arbitrary versions of the game. Each variation and version can have differences; for instance, Piglins delay for eight seconds on Bedrock, rather than six seconds on Java. Version 1.16.1 gave you a $\frac{20}{423}$ chance of an Ender pearl barter, while version 1.16.2 gave you a $\frac{10}{459}$ chance. This analysis assumes the Java variation, version 1.16.1, is being run.

Having said that, the simulation within the PE report has a blind spot. First, consider Blaze rods. In order to earn them in Minecraft, you must kill Blazes. While it is possible to kill multiple Blazes at once, the only way that's practical during a speedrun is to use the sweeping edge attack of a sword against one Blaze with multiple other near-death Blazes nearby. On top of the difficulty of achieving that feat, most Minecraft speedrunners never craft a sword. Axes do more damage per strike, are effectively a requirement to gather wood, and some speed running stategies never ask the player to attack groups. Nearly always, then, Minecraft runners kill one Blaze at a time, and thus can easily stop when they meet their quota.

A key part of a random-seed any-percent Minecraft 1.16 speedrun is earning Ender pearls via bartering with Piglins. Unlike Blaze rods, which drop instantly after a Blaze dies, Piglins only give the player an item six seconds after the player gave them a gold bar. This, combined with the low chance of earning Ender pearls via a barter, creates a major time barrier for any speedrunner. Fortunately for the player, they don't have to manually hand over every gold bar and can instead drop a stack of them at the feet of the Piglin.

Runners can speed up this process one of two ways. For their 14 minute 39 second speed run, Couriway bartered with a half-dozen Piglins simultaneously and waited until they got enough Ender pearls. By bartering with so many Piglins, though, they no longer had strict control over when to stop and thus we'd expect their barter stats to be more Binomial than Negative Binomial.

Another approach is to multitask. For their 15 minute 51 second speedrun, Dowsky trapped a Piglin in a pit, tossed gold bars into it, and while the Piglin was bartering ran off to kill Blazes. If the Blaze spawner is far enough away, the

player won't be able to see when Ender pearls drop and thus won't be able to stop bartering once their quota is met. Again, these barters would be better modelled by a Binomial than a Negative Binomial.

The simulation code present on page 17 of the PE report only models bartering with a single Piglin that the player can monitor closely. That would be perfectly fine if they could show Dream used that technique exclusively when bartering with Piglins, but the report makes no such effort.

2.2.2 Misunderstanding Minecraft

The PE report demonstrates other misunderstandings about how Minecraft is played.

> If you consider every Minecraft player, then a "perfect" ender pearl and blaze drop record (2/2 ender pearl barters and 7/7 blaze rod drops) occurs multiple times per hour, since this has a 1 in 60000 odds and Minecraft is played many millions of times a day. Considering all Minecraft worlds ever played and the multitude of ways in which luck plays a role, even one in a trillion events happen daily. (pg. 5)

I am a Minecraft player. While I haven't tracked my Ender pearl barter stats, I can state with confidence that of the zero attempts I'm made to earn Blaze rods, zero have been successful. This is because I have no interest in reaching "The End" at the moment, nor have I had the inclination to do so in the six months or so I've been playing this game. If I do get that interest, I'm playing on a private server with other people who have long since tracked down a Stronghold on our shared map and set up a minecart route to it. In practice, gathering the materials to make Eyes of Ender is something most Minecraft players rarely do, and they rarely attempt it more than once the entire time they play Minecraft. Only a very small fraction of the Minecraft community attempts to speedrun the game, and only those players routinely hunt for Ender pearls and Blaze rods. One in a trillion events can happen daily, but only if about a trillion attempts are made at those events per day.

This lack of knowledge is understandable, given the author does not appear to be a Minecraft player. Nonetheless, the more you know about a topic, the less likely your analysis of that topic will rest on false premises.

2.2.3 Lack of a Comparison to Real World Data

This premise leads the PE report to forgo comparisons to other speedrunners.

> Comparison to other runners is not necessary to establish that Dream had very low probability runs. Instead this comparison is more relevant to the interpretation of these low probabilities. For example, it reduces the plausibility that the low probabilities were due to some universal glitch that affects all speedrunners. As the reader is assessing the evidence, the low probability of Dream's runs and that Dream performed much better than other speedrunners should not be considered as independent pieces of evidence as they both are consequences of the same thing. Any lucky speedrunner chosen because they look lucky will look lucky when compared to other speedrunner streams that were chosen randomly. (pg. 15)

At the core of statistics is the idea of a reference class. By invoking every Minecraft player, the PE report is stating that Dream's performance should be compared to them, or in other words that they are the appropriate class of entities to use as a reference. Here, though, the report argues that Dream's performance should not be compared to that of other top speed runners, because it is too dissimilar; rather than compare Dream to speed runners of similar skill, the PE report states the MST report picked random speed runners.

How the MST Report chose their speedrunners.

> However, upon further research, Dream's observed drop rates are substantially greater than those of other top-level RSG runners—including, Illumina, Benex, Sizzler, and Vadikus. (pg. 3)

Setting aside the argument that this is the reverse of reality, it is notable that the report makes no attempt to compare Dream's performance to any real-world reference class. If a perfect Blaze rod and Ender pearl bartering session is indeed a common occurrence, it should be easy to demonstrate that by pointing to a few examples of ordinary Minecraft players achieving those feats. Even if they are rarely recorded, discussion of such events must appear in comment sections or Discord channel messages, and we would expect it to be common knowledge among players of Minecraft.

Conversely, even if other speedrunners are an inappropriate reference class, the comparison would help validate their methodology. The author of this report had the data to do this, thanks to the MST report, and passing it through their simulation was as easy as editing two lines of code, so there was no lack of means or opportunity.

2.2.4 Missing Context

> In this document, I don't have time to discuss the long-term debate between these different statistical paradigms and when they should be applied. The short version is that another way of investigating whether the probabilities were modified is to try to determine what probabilities were used. (pg. 5)

By skipping the details, the PE report leaves out an important consequence of using Bayesian statistics. You can simplify Bayesian stats down to

> Given the data before me, what credence do I place on a hypothesis relative to some reference?

"Credence" can be thought of in terms of "belief points:" when I say I give 100:1 odds that Joe Biden will become President of the United States, I am saying that for every one belief point I give to the hypothesis "Joe Biden will not become President" I also give one hundred towards "Joe Biden will become president."

Betting Odds

The comparison to betting odds is no accident, the study of statistics began with people trying to figure out either how to cheat at gambling or how to detect said cheating.[Gri12]

Mathematically, "credence" is the output of a likelihood function. Likelihoods obey all the same rules as probabilities, save one: while probabilities are bounded to stay between 0 and 1, likelihoods only have a lower bound of zero. The inputs to a likelihood function are the data we observe, as well as all parameters of a model capturing our belief in that data. These parameters form a multi-dimensional parameter space. All hypotheses we can make about the data live within that space. These may be point hypotheses ("Dream's Blaze rod drop rate was precisely 0.509583957"), or composite hypotheses that are a weighted subset of all possible point hypotheses ("Dream's Blaze rod drop rate ranged between 0.431 and 0.569"). If the parameter space is discrete ("which side of a six-sided die will appear"), that weighted subset is calculated by summation over all possible parameters; if it is continuous ("Dream's Blaze rod drop rate"), it is calculated by integration over all parameters.

Unfortunately, calculating those integrals directly is impractical for all but the easiest problems. Frustrated statisticians and mathematicians began looking for shortcuts and alternatives, and a number of them were codified into what we now call frequentist statistics. The p-value is one such example: it also relies on likelihood functions, but it assumes the amount of data collected is practically infinite. This allows the problem to be transformed into a simpler one that's much easier to integrate.

This has an important side-effect: the flip-side of p-values exaggerating the evidence is that Bayesian metrics are relatively conservative when given the same evidence. if the default hypothesis is some variation of "Dream played fairly", then Bayesian statistics will be slower to give up on that than frequentist statistics. To put that in bold:

Bayesian statistics is intrinsically more likely to conclude Dream played fairly, relative to frequentist statistics, given the same evidence. Whether this is a problem or not depends on how sound you think the premises are behind each system. It's not unfair to invoke p-values if the "fairer" analysis is Bayesian and you reject that system outright. By the same token, a Bayesian analysis of Dream's performance is not biased towards him relative to one that uses p-values, provided you think p-values should not be applied to this problem.

2.2.5 Biased Priors

You can think of a prior as a likelihood function supplied with zero evidence. It is convenient to define your likelihood function as how your credence changes given one atom of data, because then you can chain together arbitrary numbers of likelihood functions to handle arbitrary amounts of evidence. That chain has to be anchored somewhere, though, so you also need to define the likelihood of a specific choice of parameters absent any evidence. The opposite end of that chain is the posterior, or your credence after accounting for all the evidence.

Some priors are hypothesis-specific. "What are the odds that Dream cheated?" depends on "what are the odds of a speedrunner cheating?". If cheating is common in the speedrunning community, then the odds of a specific speedrunner cheating are high, absent any evidence that could change the odds. Likewise, the odds of a specific speedrunner cheating are low if speedrunners in general rarely cheat, again absent any evidence leading us to one conclusion or another.

This shades our view of the evidence: if we assume cheating is rare, our credence that a specific player cheated remains quite low after seeing one atom of evidence that suggests cheating. Importantly, that amount of credence is less than if we had instead assumed cheating is common. Relative to our prior credence of rare cheating, though, that one atom has shifted our credence more than it would have had we assumed cheating was common. In short, new evidence consistent with what you expected doesn't change your credence much; new evidence that isn't, does.

Some priors apply to all relevant hypotheses. A cheater is much less likely to set their Ender pearl drop rate to 100% than they are to set it to 25%. This is also true for a fair player. These two hypotheses are not perfect inverses, but they both share an assumption that extremely good luck is unlikely.

> For those savvy in Bayesian statistics, I use a flat/uniform/tophat prior on the probability boost from 1 to 5 and confirmed that these limits do not significantly affect the interpretation. In this case, this just means calculating the likelihoods on a grid from 1 to 5 and, since the prior is flat, these are equivalent to the relative posterior probabilities. This prior does not include any corrections for biases or any opinion that Dream modified his probabilities. (pg 10)

The prior used in the PE report gives as much credence to Dream earning 20 pearls after 423 barter as it does him earning 100. This is going to exaggerate the likelihood that Dream cheated in the resulting analysis.

2.3 Response to the Photoexcitation Report

The Minecraft Speedrunning Team has responded to the PE report. The best critique it makes mirrors one of mine.

> Moreover, their estimation of 300 livestreamed runs per day over the past year is highly implausible. Many runs are not livestreamed, and the estimation is based on current numbers, even though Minecraft speedrunning has grown massively in the recent months.
>
> At the time of Dream's run, there were 487 runners who had times in 1.16 - far under 1000 - and the vast majority of these were unpopular or did not stream. Selection bias could only be induced from observed runners, so speedrunners who had no significant viewership watching their attempts should not be included. (pg. 3)

The remaining critiques do not apply to the ones I have of the PE report, and do not need to be discussed here.

CHAPTER

THREE

METHODOLOGY

Given the flaws in both reports' methodology, it is fair to ask what a better one looks like.

3.1 The Binomial and Negative Binomial Distributions

As the MST report shows, when a Blaze is killed it almost always drops either zero or one Blaze rod. Both outcomes are equally likely. These drops behave like a Bernoulli process:

1. Some algorithm or thing will "emit" one of two possible outputs, canonically labelled 0 and 1.
2. The choice of which value to emit cannot be predicted beforehand.
3. The current output does not depend on what's been generated before or will be generated later.
4. As the number of values emitted becomes arbitrarily large, the ratio of zeros to ones comes arbitrarily close to a fixed value. That value does not change if we discard an arbitrary number of those outputs.

We can easily simulate that behaviour. Suppose I decided to collect Blaze rods for the first time, and set out to kill six Blazes. Since you can learn little from a single experiment, I do another four Blaze killing sprees.

```
# Execute this cell to install all dependencies locally
!pip -q install --user mpmath myst_nb numpy pandas matplotlib scipy
```

```
from fractions import Fraction

from math import log,factorial
import matplotlib.pyplot as plt
from mpmath import mp
from myst_nb import glue

import numpy as np

import pandas as pd
from scipy.stats import beta,binom,nbinom

dpi         = 200  # change this to increase/decrease the resolution charts are made␣
↪at
book_output = True # changing this to False will make it easier to view plots in a␣
↪Jupyter notebook

def fig_show( fig, name ):
    """A helper to control how we're displaying figures."""
    global book_output
```

(continues on next page)

(continued from previous page)

```
    if book_output:
        glue( name, fig, display=False )
        plt.close();
    else:
        fig.show()
```

```
# fixing the seed allows for perfect replication
random  = np.random.default_rng(8)

n = 6      # number of Blazes killed
c = 5      # number of runs to kill n Blazes

for experiment in range(c):
    results = random.choice( 2, size=n )
    [print("[Blaze rod] ", end='') if x==1 else print("[ nothing ] ", end='') for x␣
↪in results]
    print( f"   total = {sum(results)}/{n}" )
```

```
[Blaze rod] [ nothing ] [ nothing ] [Blaze rod] [ nothing ] [ nothing ]    total = 2/6
[Blaze rod] [Blaze rod] [Blaze rod] [Blaze rod] [ nothing ] [ nothing ]    total = 4/6
[Blaze rod] [ nothing ] [ nothing ] [ nothing ] [ nothing ] [ nothing ]    total = 1/6
[Blaze rod] [ nothing ] [Blaze rod] [ nothing ] [Blaze rod] [ nothing ]    total = 3/6
[ nothing ] [ nothing ] [ nothing ] [ nothing ] [Blaze rod] [Blaze rod]    total = 2/6
```

I might calculate my odds of getting exactly three rods by multiplying the odds of getting a rod (50%) by itself three times, then multiply that by the odds of not getting a rod (50%) three times. If I carry on that logic for all other possible outcomes and sum them up, however, I find the total probability of all outcomes is $7 \cdot (\frac{1}{2})^6 = \frac{7}{64}$. Those probabilities should sum to one, if I've defined a probability distribution.

The error in my logic is that I've failed to account for multiple identical outcomes. Two of the five runs shown above have two total successes, but both earn those successes in a different way. Conversely, there's only a single way for one of my runs to earn a Blaze rod from every Blaze. To correctly calculate the probability of k drops when killing n Blazes, I need to also calculate the number of ways I could have earned those drops. As luck would have it, the math to do that is pretty easy.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

where "!" is the factorial function, $n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2$. When I combine those two components, I get the Binomial distribution.

$$\text{Binom}(n, k, p) = \binom{n}{k} p^k (1-p)^{n-k} = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}, \tag{3.1}$$

where n is the number of times the Bernoulli process is sampled, k is the number of 1's emitted, and p is the probability of that Bernoulli process emitting a 1. For my toy example above, we find the odds for $n = 6$, $k = 3$, and $p = \frac{1}{2}$ are not $\frac{7}{64}$ but instead $\frac{5}{16}$. If I carry that on for all seven values of k and sum the probabilities, they do indeed add up to one.

```
def binomial( n, k, p ):
    return Fraction(factorial(n), factorial(k)*factorial(n-k)) * (p**k) * ((1-p)**(n-
↪k))

n = 6
p = Fraction(1,2)
```

(continues on next page)

(continued from previous page)

```
total = sum( binomial(n, k, p) for k in range(n+1) )
print( f'The sum of probabilities of all possible k is {total}', end='' )
if total == Fraction(1,1):
    print( ', which means Binom(n,k,p) defines a probability distribution over k.')
else:
    print( '.' )
```

```
The sum of probabilities of all possible k is 1, which means Binom(n,k,p) defines a␣
↪probability distribution over k.
```

Suppose I make one tweak to the above scenario, though. This time around I want to earn three Blaze rods, to allow me to craft the six Ender pearls I have in my inventory into six Eyes of Ender.

```
random  = np.random.default_rng(15)

k = 3      # number of Blaze rods needed
c = 5      # number of runs to earn k Blaze rods

for experiment in range(c):
    results = random.choice( 2, size=16 )
    while np.sum(results) < k:
        results = np.append( results, random.choice( 2, size=16 ) )
    n = np.sum( np.cumsum( results ) < k ) + 1
    [print("[Blaze rod] ", end='') if x==1 else print("[ nothing ] ", end='') for x␣
↪in results[:n]]
    print( f"   total = {k}/{n}" )
```

```
[Blaze rod] [Blaze rod] [Blaze rod]    total = 3/3
[Blaze rod] [ nothing ] [ nothing ] [Blaze rod] [ nothing ] [Blaze rod]    total = 3/6
[ nothing ] [ nothing ] [ nothing ] [ nothing ] [ nothing ] [Blaze rod] [Blaze rod]␣
↪[Blaze rod]    total = 3/8
[ nothing ] [Blaze rod] [Blaze rod] [Blaze rod]    total = 3/4
[Blaze rod] [ nothing ] [ nothing ] [ nothing ] [Blaze rod] [ nothing ] [Blaze rod]  ␣
↪ total = 3/7
```

Before, I fixed the number of Blazes I killed, n, as well as the odds of earning a Blaze rod, p, so the only thing that could vary was the number of rods k. Now with k and p fixed, n must be the one to vary. That part is obvious, but look carefully at the last Blaze I kill before quitting. Each and every time, it drops a Blaze rod. In hindsight that makes perfect sense: I've got no reason to continue killing Blazes after I earn my third rod, after all. In the process, though, I've placed some constraints on the number of ways to earn three Blaze rods. The math we invoked earlier to calculate the total number of combinations no longer works.

There is a way to fix it. Rather than consider the full n kills, let's stop at the $n-1$-th kill. At this point, I have $k-1$ Blaze rods. Since this is Bernoulli process, I have no idea that my next Blaze will net me a rod and I can exit this thought experiment. The $n-1$-th Blaze may or may not have dropped a rod, as of this point all possible ways to earn $k-1$ rods via $n-1$ Blazes are available. There's only one way to earn that k-th rod on my n-th, so it doesn't contribute any new combinations and the total number is $\binom{n-1}{k-1}$ instead of $\binom{n}{k}$.

The probability of getting those rods still works, though. The odds of me getting k rods over n trials with success probability p does not change whether I stop based on n or k. Combining this all together, we find the equation that describes the odds of n trials when trying to earn k successes with probability of success p is

$$\binom{n-1}{k-1}p^k(1-p)^{n-k} = p \cdot \text{Binom}(n-1, k-1, p) = \text{NegBinom}(n, k, p) \tag{3.2}$$

This is known as the Negative Binomial distribution. While it's closely related to the Binomial distribution, it is not identical.

3.2 Conjugate Priors

Remember when I mentioned in *Missing Context* that calculating the integrals necessary for Bayesian statistics "is impractical for all but the easiest problems?" Here's one exception: let's switch out the Bernoulli process for another that emits real numbers that have a Gaussian distribution. We know the variance of this process but not the mean value. How could we estimate it?

An obvious approach is to leverage the central limit theorem: grab a bunch of numbers from this process, calculate their average, and use that to be the best estimate. That doesn't give us a degree of credence for that estimate, though, we'd prefer some sort of distribution to represent our credence for where the mean is. We can get that by generating a Gaussian distribution representing our credence for where the mean lies. It would be handy to be able to take in multiple values at once, and we can again use the central limit theorem to construct a Gaussian distribution representing the likelihood. We still need some sort of prior, but since we know the variance we can just use a diffuse Gaussian distribution based loosely on that and rely on the data to tug the posterior in the right direction.

Gaussian Processes

I'm deliberately not calling this a Gaussian process, because that's a bit like calling a 1908 Model-T Ford a "car;" while technically correct, equivocating the two fails to capture the generality and full potential of the latter.

Noting that both the prior and posterior are Gaussian distributions. You might have figured out this allows us to iteratively update our posterior: as more data comes in, we turn it into a Gaussian prior, apply the Gaussian likelihood function, and get back another Gaussian posterior. What you may not have figured out is that the math to update our posterior is trivially easy:

$$\sigma^2_{\text{posterior}} = \left(\frac{1}{\sigma^2_{\text{prior}}} + \frac{n}{\sigma_n} \right)^{-1}$$

$$\mu_{\text{posterior}} = \sigma^2_{\text{posterior}} \left(\frac{\mu_{\text{prior}}}{\sigma^2_{\text{prior}}} + n\frac{\mu_n}{\sigma^2_n} \right),$$

where μ_n and σ_n are the mean and standard devation of $\{x_1, x_2, \ldots, x_n\}$ new observations.

When the prior and posterior of a model follow the same distribution, that prior is called a conjugate prior. These are oases of easy math in what is otherwise a Bayesian wasteland of tangled computations. If we can find a conjugate applicable to the Binomial and Negative Binomial, our analysis will be immensely simpler. The Gaussian example I used above is tempting, as the Gaussian distribution is often used as an approximation of the Binomial.

But there's no need to approximate here: the Binomial distribution has a conjugate prior, the Beta distribution. Even better, the appropriate updating rule doesn't require a fixed k or n. This saves us from having to apply corrections for speed-runners that choose different Blaze rod quotas or happen to pick up Ender pearls outside of bartering. We will need to do some math to convert it to handle the Negative Binomial scenario, though.

The updating rule for the Beta conjugate prior is

$$\text{Binom}(n, k, p) \cdot \text{Beta}(\alpha_{\text{prior}}, \beta_{\text{prior}}, p) \rightarrow \text{Beta}(\alpha_{\text{prior}} + k, \beta_{\text{prior}} + n - k, p) \tag{3.3}$$

$$\text{Beta}(\alpha, \beta, p) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} p^{\alpha-1}(1-p)^{\beta-1} \tag{3.4}$$

where $\Gamma(x)$ is the Gamma function. It is an extrapolation of factorials, such that $n! = \Gamma(n+1)$ for any positive integer n. Unlike factorials, though, the Gamma function can be applied to fractions, real numbers, and even complex numbers.

Now we need to decide which Beta distribution to use as a prior. *Two options are commonly used*, the Bayes/Laplace prior ($\alpha = \beta = 1$) and the Jeffreys prior ($\alpha = \beta = \frac{1}{2}$). The PE report uses the Bayes/Laplace prior, and as per

my critique in *Biased Priors* I do not think it is appropriate here. The same reasoning applies to the Jeffreys prior; it groups the credence around $p = 0$ and $p = 1$, both of which are unlikely possibilities in the context of Blaze rod drops and Ender pearl barters.

```
fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

x = np.linspace(0,1,512)
plt.plot( x, beta.pdf(x, 1, 1), '-r', label='Bayes/Laplace Prior')
plt.plot( x, beta.pdf(x, .5, .5), '-b', label="Jeffrey's Prior" )

plt.ylabel("Likelihood")
plt.ylim([0,3])
plt.yticks([])

plt.xlabel("p")
plt.xticks([0, .5, 1])

plt.legend()
fig_show( fig, "fig:common_priors" )
```

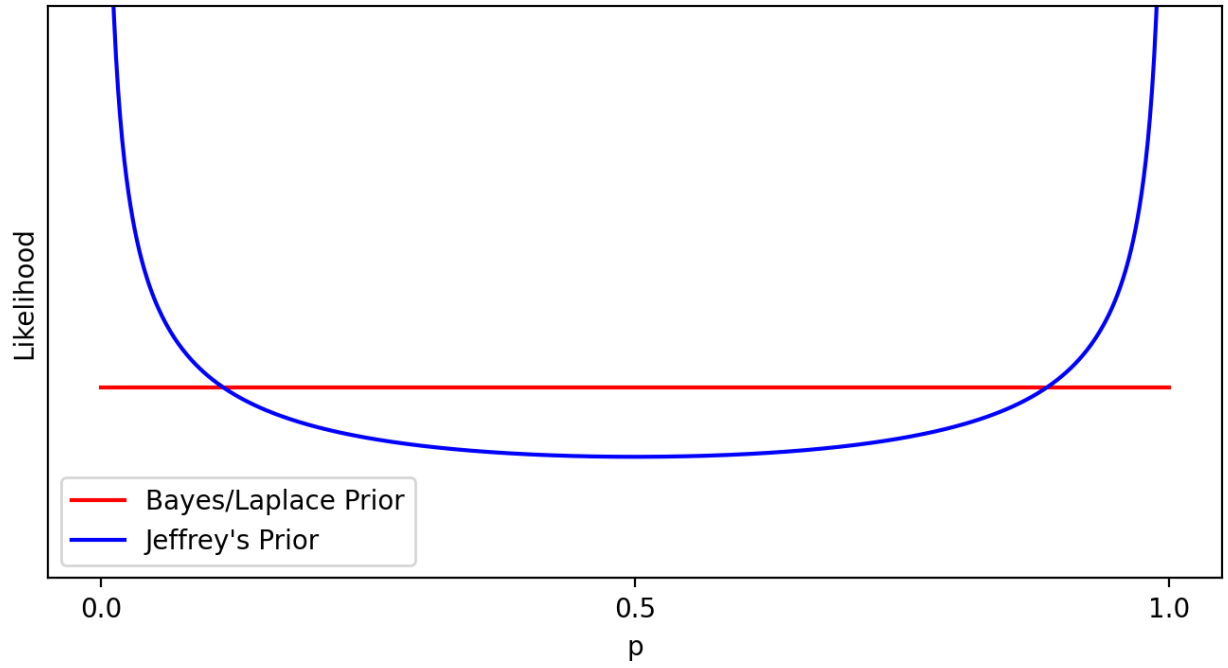

Fig. 3.1: The two most commonly used priors for the Beta distribution.

The prior I recommend instead places the bulk of the credence around the expected drop rate for each item, and spreads the remainder out according to a "strength" factor.

```
def prior(r, strength):
    """Generate an appropriate prior to be used during inference with a Beta␣
↪conjugate prior.

    Parameters
    ----------
    r = The expected rate of item drops, assuming no modification of the game. Must␣
↪be between 0 and 1.
```

(continues on next page)

(continued from previous page)

```
    strength = A value to strengthen our credence around r. Higher values favour the␣
↪hypothesis that
        someone played fairly, while lower values favour someone who cheated. Must be␣
↪a positive real.

    Returns
    -------
    A tuple (alpha, beta) representing the Beta prior."""

    # ensure the proper values are given
    assert (r > 0) and (r < 1)
    assert strength > 0

    if r < .5:
        return (strength, strength*(1-r)/r)
    else:
        return (strength*(1-r)/r, strength)
```

```
fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')
(ax1, ax2) = fig.subplots(1, 2)

x = np.linspace(0,1,512)

strengths = [2,4,8]
labels = [f'strength={strength}' for strength in strengths]
r_blaze = Fraction(1,2)
r_pearl = Fraction(20,423)

for i,strength in enumerate(strengths):

    blaze_prior = [float(x) for x in prior( r_blaze, strength )]
    ax1.plot(x, beta.pdf(x, *blaze_prior), '-', alpha=0.6, label=labels[i] )

    pearl_prior = [float(x) for x in prior( r_pearl, strength )]
    ax2.plot(x/5, beta.pdf(x/5, *pearl_prior), '-', alpha=0.6, label=labels[i] )


ax1.legend()
ax1.set_xticks([0,.5,1])
ax1.set_xlabel('drop rate, rods')
ax1.set_yticks([])
ax1.set_ylabel('likelihood')
ax1.axvline( r_blaze )

ax2.legend()
ax2.set_xticks([0,float(r_pearl),.1,.2])
ax2.set_xlabel('barter rate, pearls')
ax2.set_yticks([])
ax2.axvline( r_pearl )

fig_show( fig, "fig:sub_priors")
```

These priors are all quite subjective, and ask you to make a judgment call on the appropriate strength value to use. I think 4 is a good compromise, but I'll keep α_{prior} and β_{prior} as free parameters. If you disagree with my subjective Beta prior, you'll be free to substitute in any other. To double-check the effect of the strength parameter, I'll also do a sensitivity analysis with the real-world data to get a better handle on the prior's influence.

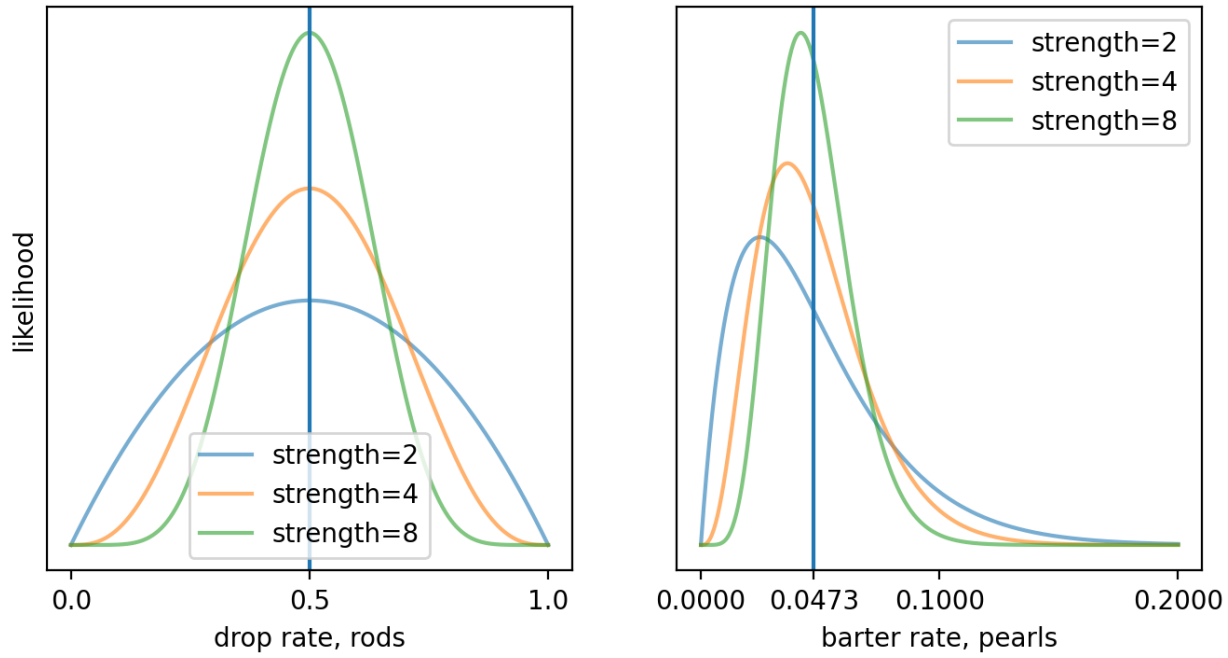

Fig. 3.2: Comparing my subjective priors for various strength factors. The vertical line is the expected or average value of the Bernoulli process.

The apparent skew with the Ender pearl prior is because we are fixing the mean instead of the maximal likelihood. As we'll be integrating across this prior, fixing the latter would bias us to assume the barter rate is higher.

3.3 The Posterior Distribution for the Binomial and Negative Binomial

We can now construct a posterior distribution for both the Binomial and Negative Binomial cases.

The updating rule of Equation (3.3) only works for one run of k successes via n trials, distributed according to the Binomial, but it is simple enough to extend for multiple runs. Define $\vec{n} = \{n_1, n_2, \ldots, n_c\}$ as a vector that contains the trial counts for all c runs, and $\vec{k} = \{k_1, k_2, \ldots, k_c\}$ as a vector for each success count. The resulting Beta posterior is

$$\text{Beta}(\alpha_{\text{prior}} + \sum_{j=1}^{c} k_j, \beta_{\text{prior}} + \sum_{j=1}^{c}(n_j - k_j), p) = \frac{\Gamma\left(\alpha_{\text{prior}} + (\sum_{j=1}^{c} k_j) + \beta_{\text{prior}} + (\sum_{j=1}^{c}(n_j - k_j))\right)}{\Gamma(\alpha_{\text{prior}} + \sum_{j=1}^{c} k_j)\Gamma(\beta_{\text{prior}} + \sum_{j=1}^{c}(n_j - k_j))} p^{\alpha_{\text{prior}}-1+\sum_{j=1}^{c} k_j}(1-p)^{\beta_{\text{prior}}-1+} \tag{3.5}$$

Equation (3.5) is a bit ugly to look at. One way to clean it up is to use different notation. The Taxicab norm is defined as

$$\|\vec{n}\|_1 = \sum_{j=1}^{c} |n_j| \tag{3.6}$$

Since no value in $\vec{n}$ or $\vec{k}$ is negative, taking the absolute value has no effect. Note as well that

$$\sum_{j=1}^{c}(n_j - k_j) = \left(\sum_{j=1}^{c} n_j\right) - \left(\sum_{j=1}^{c} k_j\right) = \|\vec{n}\|_1 - \|\vec{k}\|_1, \tag{3.7}$$

which, along with $\|\vec{k}\|_1 + \|\vec{n}\|_1 - \|\vec{k}\|_1 = \|\vec{n}\|_1$, allows us to rewrite Equation (3.5) as

$$\text{Beta}(\alpha_{\text{prior}} + \|\vec{k}\|_1, \beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1, p) = \frac{\Gamma(\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1)}{\Gamma(\alpha_{\text{prior}} + \|\vec{k}\|_1)\Gamma(\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1))} p^{\alpha_{\text{prior}} + \|\vec{k}\|_1 - 1}(1-p)^{\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1 - 1)} \tag{3.8}$$

There's no need to normalize Equation (3.8) to make it a probability distribution over p, as

$$\int_{p=0}^{1} \text{Beta}(\alpha, \beta, p) = 1 \tag{3.9}$$

Or, in English, it was already normalized over p.

That handles the case when $\vec{k}$ and $\vec{n}$ are taken from a Binomial distribution. What about the Negative Binomial case? Equation (3.2) shows the Negative Binomial is simply the Binomial times the probability of success. In theory, we could treat it the same as the Binomial case and apply some sort of correction to align it with the Negative Binomial. We have to be careful, though, as that equation also shows the Negative Binomial isn't a probability distribution with respect to p.

```
n = 6  # integrates to one with respect to p implies it does so regardless of what k
↪and n are.
k = 3

# mp.quad() uses quadrature to numerically integrate. No math necessary on our side!
result = mp.quad( lambda p: p*binomial(n, k, p), (0,1) )
glue( 'nbinom_integral', float(result), display=False )

print( f'The integral of all possible p, for n={n} and k={k}, is {result}', end='' )
if result == Fraction(1,1):
    print( ', which means p*Binom(n,k,p) defines a probability distribution over p
↪for at least that (n,k) pair.')
else:
    print( '.' )
```

```
The integral of all possible p, for n=6 and k=3, is 0.0714285714285714.
```

It is actually a probability distribution with respect to n. We can fix that by normalizing it, and the best time to do so is after extending the updating rule. First, observe that

$$\sum_{j=1}^{c}(n_j - 1) = \left(\sum_{j=1}^{c} n_j\right) - c = \|\vec{n}\|_1 - c \tag{3.10}$$

$$(\|\vec{n}\|_1 - c) - (\|\vec{k}\|_1 - c) = \|\vec{n}\|_1 - \|\vec{k}\|_1, \tag{3.11}$$

and secondly, remember that multiplication is commutative. This allows us to rearrange terms and write

$$\text{posterior}_{\text{Negative Binomial}} = p^c \left(\left(\prod_{j=1}^{c} \text{Binom}(n_j - 1, k_j - 1, p)\right) \text{Beta}(\alpha_{\text{prior}}, \beta_{\text{prior}}, p)\right) \tag{3.12}$$

$$= p^c \text{Beta}(\alpha_{\text{prior}} + \|\vec{k}\|_1 - c, \beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1, p) \tag{3.13}$$

$$= p^c \frac{\Gamma(\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1 - c)}{\Gamma(\alpha_{\text{prior}} + \|\vec{k}\|_1 - c)\Gamma(\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1)} p^{\alpha_{\text{prior}} + \|\vec{k}\|_1 - c - 1}(1-p)^{\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1 - 1} \tag{3.14}$$

We can tidy that up a bit, as $p^a \cdot p^b = p^{a+b}$ and $c - c = 0$.

$$= \frac{\Gamma(\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1 - c)}{\Gamma(\alpha_{\text{prior}} + \|\vec{k}\|_1 - c)\Gamma(\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1)} p^{\alpha_{\text{prior}} + \|\vec{k}\|_1 - 1}(1-p)^{\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1 - 1} \tag{3.15}$$

To normalize Equation (3.15) according to p, we only need to divide it by what it integrates to across p.

$$\int_{p=0}^{1} \frac{\Gamma(\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1 - c)}{\Gamma(\alpha_{\text{prior}} + \|\vec{k}\|_1 - c)\Gamma(\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1)} p^{\alpha_{\text{prior}} + \|\vec{k}\|_1 - 1}(1-p)^{\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1 - 1} \tag{3.16}$$

Note that all the Gamma functions are constants, unaffected by any change in p. As $\int_x c \cdot g(x) = c \int_x g(x)$, we can pull them out of the integral. Since all the Gamma functions are identical across the numerator and denominator, they cancel out and we are left to evaluate

$$\int_{p=0}^{1} p^{\alpha_{\text{prior}} + \|\vec{k}\|_1 - 1}(1-p)^{\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1 - 1} \tag{3.17}$$

While this may seem daunting, remember that the Beta distribution integrates to 1 across p. Thus we can rearrange Equation (3.9) to find

$$\int_{p=0}^{1} \text{Beta}(\alpha, \beta, p) = 1 \tag{3.18}$$

$$\int_{p=0}^{1} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} p^{\alpha-1}(1-p)^{\beta-1} = 1 \tag{3.19}$$

$$\int_{p=0}^{1} p^{\alpha-1}(1-p)^{\beta-1} = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \tag{3.20}$$

Equation (3.20) will be incredibly handy in future, but for now it allows us to finish normalizing Equation (3.15).

$$\alpha - 1 = \alpha_{\text{prior}} + \|\vec{k}\|_1 - 1 \tag{3.21}$$

$$\alpha = \alpha_{\text{prior}} + \|\vec{k}\|_1 \tag{3.22}$$

$$\beta = \beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1 \tag{3.23}$$

$$\text{posterior}_{\text{Negative Binomial}} = \frac{p^{\alpha_{\text{prior}} + \|\vec{k}\|_1 - 1}(1-p)^{\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1 - 1}}{\frac{\Gamma(\alpha_{\text{prior}} + \|\vec{k}\|_1)\Gamma(\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1)}{\Gamma(\alpha_{\text{prior}} + \beta_{\text{prior}} \|\vec{n}\|_1)}} \tag{3.24}$$

$$= \frac{\Gamma(\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1)}{\Gamma(\alpha_{\text{prior}} + \|\vec{k}\|_1)\Gamma(\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1)} p^{\alpha_{\text{prior}} + \|\vec{k}\|_1 - 1}(1-p)^{\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1 - 1} \tag{3.25}$$

Proving the Binomial Update Rule

This is also why I didn't bother showing the derivation of the Binomial update rule: calculating $\text{Binom}(n, k, p) \cdot \text{Beta}(\alpha, \beta, p)$ is trivially harder than $p \cdot \text{Beta}(\alpha, \beta, p)$, and the renormalization process is exactly the same. The extension from k and n to $\vec{k}$ and $\vec{n}$ is also trivial.

This is identical to Equation (3.8). The PE report was correct to raise the possibility that barters and drops do not follow the Binomial distribution (pg. 7), but an analysis that used the Negative Binomial instead would come to the same conclusion. The MST report is also incorrect when they state that the Binomial distribution is "a very good approximation" (pg. 10) for barters/drops, in reality it leads to the exact same outcome as assuming the Negative Binomial instead.

The posterior from Equation (3.25) is worth seeing in action. We'll apply it to the last toy example from *The Binomial and Negative Binomial Distributions*, using the prior I defined earlier.

```
random  = np.random.default_rng(15)

k = 3      # number of Blaze rods needed
c = 5      # number of runs to earn k Blaze rods
```

(continues on next page)

(continued from previous page)

```
sum_k = k * c  # we already know this
sum_n = 0

for experiment in range(c):
    results = random.choice( 2, size=16 )
    while np.sum(results) < k:
        results = np.append( results, random.choice( 2, size=16 ) )
    sum_n += np.sum( np.cumsum( results ) < k ) + 1

my_prior = prior(0.5, 4)

fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

x = np.linspace(0,1,512)
plt.fill_between( x, beta.pdf( x, my_prior[0] + sum_k, my_prior[1] + sum_n - sum_k),␣
↪linestyle='-',
                 label=f'prior + (sum_n = {sum_n}, sum_k = {sum_k})')

plt.plot( x, beta.pdf( x, my_prior[0], my_prior[1]), '-k', \
          label=f'prior (Beta({my_prior[0]:.1f},{my_prior[1]:.1f}))')
plt.axvline( 0.5, color='red' )

plt.xticks([0,.5,1])
plt.xlabel('p')

plt.yticks([])
plt.ylabel('likelihood')

plt.legend()
fig_show( fig, "fig:posterior_in_action")
```

The results are as expected. Our credence over the Blaze rod drop rate was fairly diffuse to begin with. Adding the simulation data grouped it more tightly together, making extremely low or high drop rates very unlikely. At the same time, the limited evidence allowed drop rates close to $p = 0.5$ to remain very credible, and random variation kept the most likely drop rate of the posterior from matching the actual value. We're more certain of the rate, but there's still some wiggle room.

3.4 Defining Fairness

That last paragraph reveals a big problem. We have a mathematically-precise posterior distribution that contains our credence for all possible Blaze rod drop rates, but we're not equally interested in all of them. Instead, we're trying to translate the fuzzy concept of "fairness" into some subset of those rates and eyeballing how those specific rates perform within the posterior. The eyeballing part simply won't do, as it doesn't lay out all the underlying assumptions and relies on an estimate generated by a human being. We need to be much more rigorous, and come up with a mathematical definition of "did Dream cheat?" There is no obvious and unambiguous definition, but some definitions are better than others.

The easiest way to start is not by defining cheating, but instead defining fair play. We know unaltered Minecraft code sets the drop rate to 0.5, so why not define fairness by Beta($\alpha_{\text{posterior}}, \beta_{\text{posterior}}, 0.5$)?

```
print( 'The likelihood I was playing fair in my simulation is' +
      f' {beta.pdf( 0.5, my_prior[0] + sum_k, my_prior[1] + sum_n - sum_k)} .')
```

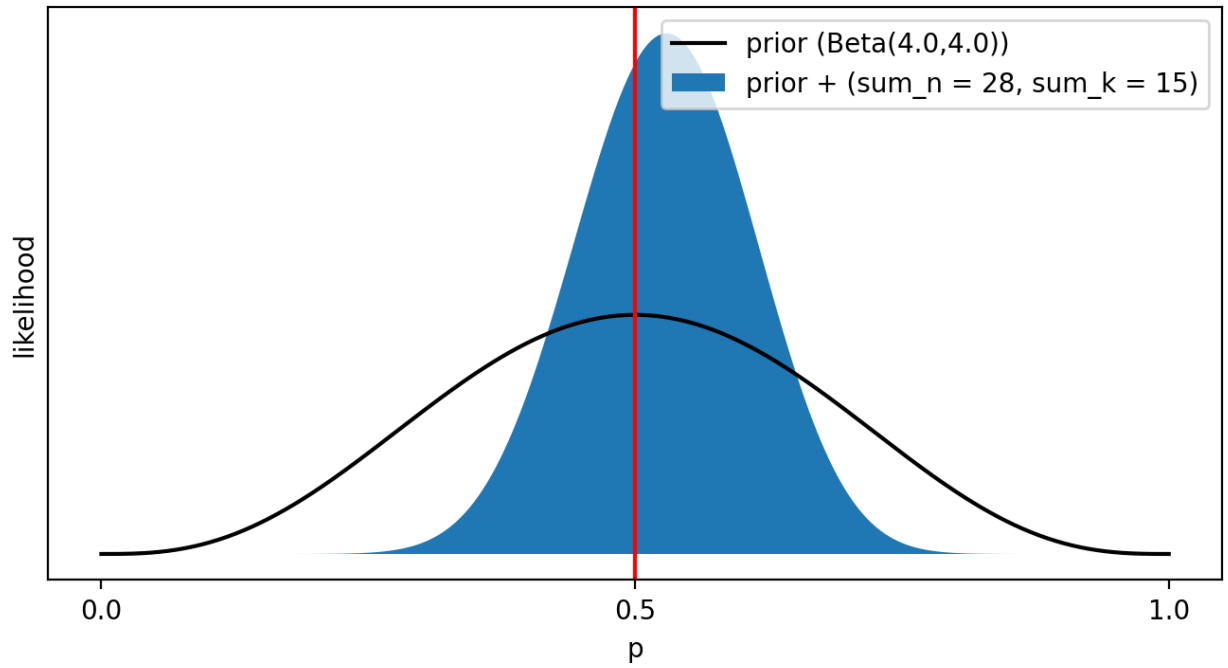

Fig. 3.3: The posterior for the last toy example of *The Binomial and Negative Binomial Distributions*, generated after adding the data to the previously-defined prior. The red vertical line coincides with the true drop rate.

```
The likelihood I was playing fair in my simulation is 4.490060385433022 .
```

That won't work, if only because the output of the posterior is a likelihood, which isn't normalized like a probability is. We need to find some way to scale or modify that value so it maps to something more meaningful to us.

But there's a more subtle problem here. Imagine an idealized dart board, centred at the origin of a 2D Cartesian plane and with radius 1. Now pick a point on that dart board, and ask yourself the odds of an ideal dart hitting that point. The answer must be zero: since the dart board contains an uncountably infinite number of points, the odds of a dart hitting any one point is zero. Since the point we chose was arbitrary, we must conclude the odds of an ideal dart hitting our ideal dart board are zero. And yet clearly an ideal dart cast at that board must hit some point.

In real life, we never work with infinitely small points. Instead, the tip of the dart has a small but finite area, as well as the area we pick on the board. When we switch the idealized case to match, the probability of hitting the board becomes non-zero. Likewise, I could collect Blaze rods in Minecraft until the day I die, and the odds of me getting an exact 50% drop rate would be incredibly small even if I was playing with an unmodified version of Minecraft. Instead, when assessing fairness we should consider a range of possible drop rates. Where do we place the ends of the range, however?

One possibility comes from coin flipping. We've considered that to be a fair random process for a long time, to the point that it's used to settle tied elections. One researcher decided to test how fair a coin toss actually is, by asking medical residents to try biasing their coin flips in order to earn a $20 prize. They were given a few weeks' notice, a few minutes of practice to warm up, and then asked to flip a coin 300 times.

> This study shows that when participants are given simple instructions about how to manipulate the toss of a coin and only a few minutes to practise this technique, more than half can significantly manipulate the outcome. With devoted training, more participants would probably be able to achieve this figure, and the magnitude of the manipulation would probably be increased.[CW09]

If it is so easy to bias a coin toss, how did it get the reputation of being unbiased? Perhaps a coin toss is "fair enough," in that the amount of bias you can generate isn't enough to significantly change the outcome. If we take the average

bias of coin tosses in this study as our threshold, then "fair enough" constitutes no more than an $\frac{569}{500}$ improvement in the odds over perfect fairness, or in this case a success rate of 56.9%. We could express that in mathematics as

$$\int_{p=0.431}^{0.569} \text{Beta}(\alpha_{\text{posterior}}, \beta_{\text{posterior}}, p) \tag{3.26}$$

This definition also has problems, though. Suppose I decided to modify my copy of Minecraft so that Blaze rods dropped 53% of the time. In the short run this is quite undetectable, but watch what happens when I target seven Blaze rods across 300 sessions.

```
random  = np.random.default_rng(26)

k       = 7     # number of Blaze rods needed
c       = 300   # number of runs to earn k Blaze rods
r_cheat = 0.53  # the true drop rate of Blaze rods

sum_k = k * c  # we already know this
sum_n = sum_k + np.sum(random.negative_binomial( k, r_cheat, size=c ))
glue( 'deffair_number_blazes_killed', sum_n, display=False )

my_prior = prior(0.5, 4)

fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

x = np.linspace(.45,.6,512)
plt.fill_between( x, beta.pdf( x, my_prior[0] + sum_k, my_prior[1] + sum_n - sum_k),␣
↪linestyle='-',
                 label=f'prior + (sum_n = {sum_n}, sum_k = {sum_k})')

plt.axvline( 0.431, color='red' )
plt.axvline( 0.569, color='red' )

plt.xticks([.431, .5, r_cheat, .569])
plt.xlabel('p')

plt.yticks([])
plt.ylabel('likelihood')

plt.legend()
fig_show( fig, 'fig:fixed_fairness_bad' )
```

The posterior appears to provide strong evidence that I altered my drop rate, as little credence is massed around $p = \frac{1}{2}$. Yet almost all of the credence lies within the bounds of integration, so our calculated value is effectively identical to a perfectly fair Blaze rod drop rate.

The 56.9% definition of "fair enough" is for coin tosses, which are rarely done more than once or twice in a sitting. Blaze rod drops can be easily modelled by coin flips, and yet *the above figure* tells me I performed 4028 total flips. Our definition of "fair enough" depends on what we can detect, and the more events we observe the better we can detect subtle biases. It isn't enough for H_{fair} to integrate over a range, its must "tighten up" as more data arrives.

Consider a version of Minecraft that swapped out the pseudo-random number generator with a sophisticated non-random algorithm. The game can detect the number of Blaze rods I want, and automatically ensure exactly half of the Blazes I kill will drop Blaze rods. If I fed this fair algorithm into the math and generated a posterior, for the $\|\vec{n}\|_1 = 28$ case I would get

```
random  = np.random.default_rng(15)

k = 3     # number of Blaze rods needed
```

(continues on next page)

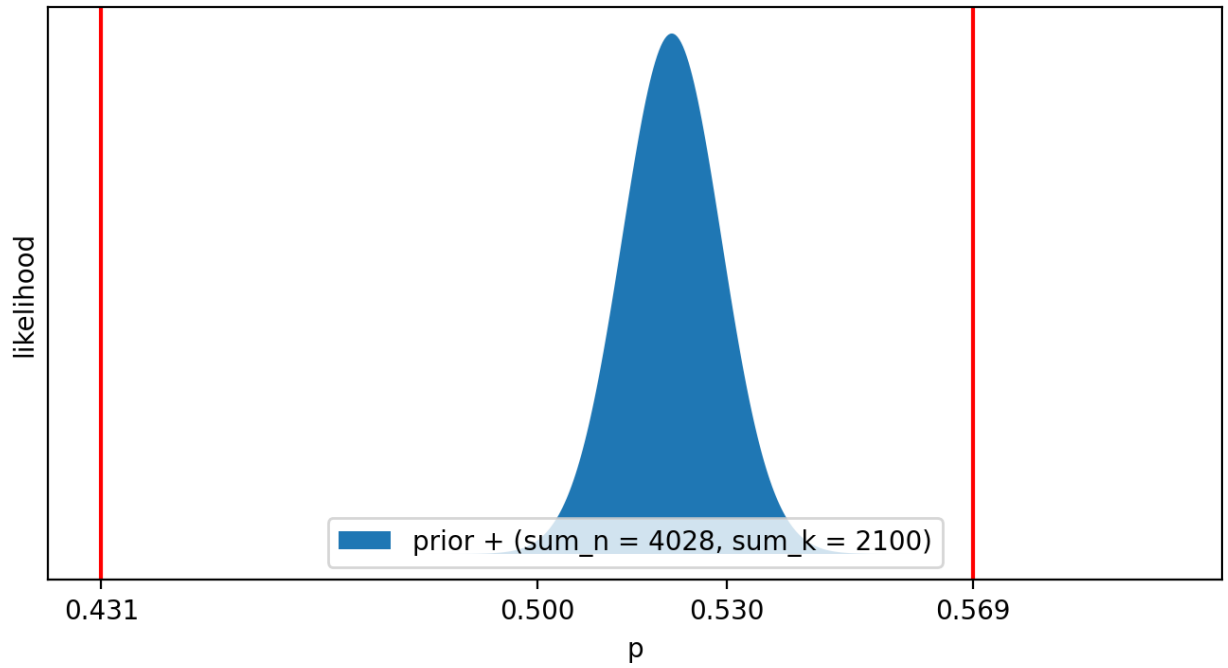

Fig. 3.4: The posterior when I target seven Blaze rods over the course of 300 runs, with a 53% chance of success.

(continued from previous page)

```
c = 5      # number of runs to earn k Blaze rods

sum_k = k * c  # we already know this
sum_n = 0

for experiment in range(c):
    results = random.choice( 2, size=16 )
    while np.sum(results) < k:
        results = np.append( results, random.choice( 2, size=16 ) )
    sum_n += np.sum( np.cumsum( results ) < k ) + 1

my_prior = prior(0.5, 4)

fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

x = np.linspace(0,1,512)
plt.fill_between( x, beta.pdf( x, my_prior[0] + sum_k, my_prior[1] + sum_n - sum_k),␣
↪linestyle='-',
                 label=f'prior + (sum_n = {sum_n}, sum_k = {sum_k})')

plt.plot( x, beta.pdf( x, my_prior[0] + sum_n/2, my_prior[1] + sum_n/2), '-k', \
          label=f'fair play (Beta({my_prior[0] + sum_n/2:.1f},{my_prior[1] + sum_n/2:.
↪1f}))')
plt.axvline( 0.5, color='red' )

plt.xticks([0,.5,1])
plt.xlabel('p')

plt.yticks([])
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('likelihood')

plt.legend()
fig_show( fig, 'fig:fair_play' )
```

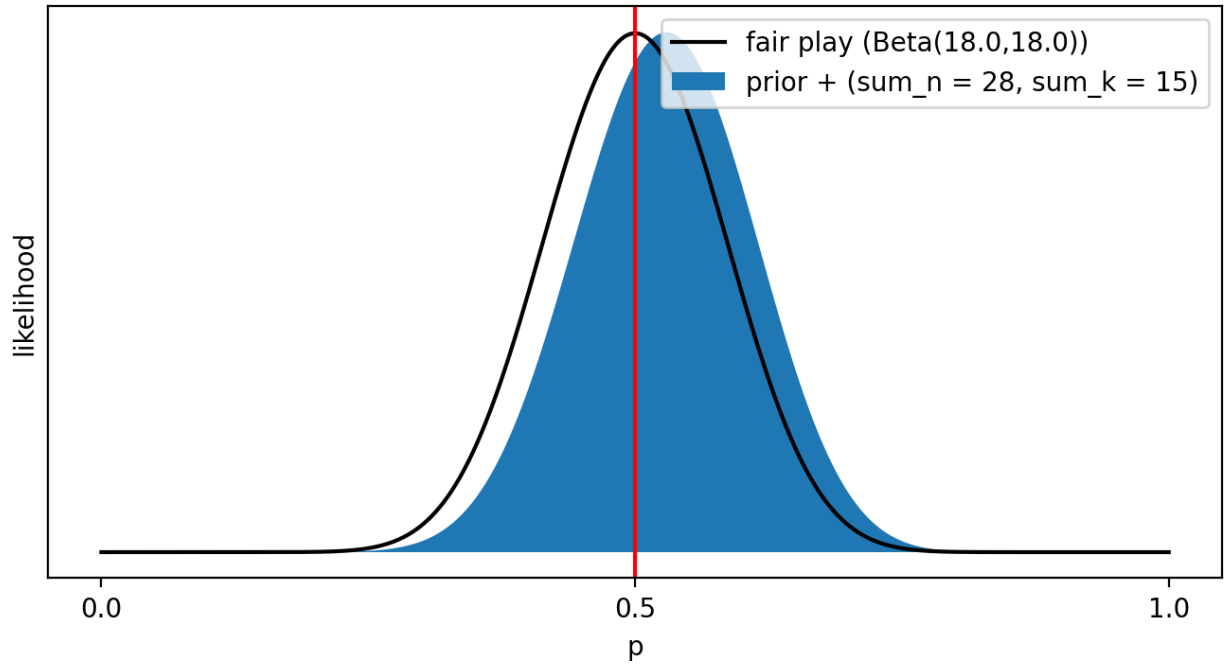

Fig. 3.5: The posterior after a perfectly fair play session, compared to the posterior for the last toy example of *The Binomial and Negative Binomial Distributions*.

I consider *this posterior* to be a really good match for H_{fair}. You might have thought of a problem, though: what if I'd killed twenty-seven Blazes instead of twenty-eight? Or, for a more extreme example, a perfectly fair Ender pearl barter session is twenty successes for every 423 attempts. Those two numbers are co-prime, so the number of successful pearl barters will only be an integer if the number of attempts is a multiple of 423. It would be terribly inconvenient to force players to barter exactly 423 times so we can analyze their bartering rate.

Thankfully for us, the Gamma function is defined for fractions. For Ender pearl barters, we can construct H_{fair} by assigning $\frac{20}{423}$ successes for every single barter, instead of twenty successes for every 423 barters. Now we can handle an arbitrary number of drops or barters.

Let's lock the mathematics for $H_{\text{fair}}(\|\vec{n}\|_1)$ down.

$$H_{\text{fair}}(p \mid \|\vec{n}\|_1) = \text{Beta}(\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1, \beta_{\text{prior}} + (1 - r_{\text{fair}})\|\vec{n}\|_1, p), \tag{3.27}$$

where r_{fair} is the rate we expect during fair play. For instace, $r_{\text{blaze}} = \frac{1}{2}$ and $r_{\text{pearl}} = \frac{20}{423}$.

If you do not agree with my definition of "fair play" according to Equation (3.27), I'd encourage you to think up your own and express it in a mathematically precise way. In the meantime, I'll use my definition to create a likelihood from the posterior. To do that, I multiply $H_{\text{fair}}(\|\vec{n}\|_1)$ by the Beta posterior and integrate the combination.

$$\int_{p=0}^{1} \text{Beta}(\alpha_{\text{posterior}}, \beta_{\text{posterior}}, p) H_{\text{fair}}(\vec{n}, p) = \int_{p=0}^{1} \text{Beta}(\alpha_{\text{prior}} + \|\vec{k}\|_1, \beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1, p) \cdot \text{Beta}(\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1, \beta_{\text{prior}} + (1 - r_{\text{fair}})\|\vec{n}\|_1 \tag{3.28}$$

$$= \int_{p=0}^{1} \frac{\Gamma(\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1)}{\Gamma(\alpha_{\text{prior}} + \|\vec{k}\|_1)\Gamma(\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1)} p^{\alpha_{\text{prior}} + \|\vec{k}\|_1 - 1}(1-p)^{\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1 - 1} \frac{\Gamma(\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1)}{\Gamma(\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1)\Gamma(\beta_{\text{prior}} + (1 - r_{\text{fair}})\|\vec{n}\|_1)} \tag{3.29}$$

$$= \frac{\Gamma(\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1)^2}{\Gamma(\alpha_{\text{prior}} + \|\vec{k}\|_1)\Gamma(\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1)\Gamma(\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1)\Gamma(\beta_{\text{prior}} + (1 - r_{\text{fair}})\|\vec{n}\|_1)} \int_{p=0}^{1} p^{2\alpha_{\text{prior}} + \|\vec{k}\|_1 + r_{\text{fair}}\|\vec{n}\|_1 - 2}(1-p)^{2\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1 + (2 - r_{\text{fair}})\|\vec{n}\|_1 - 2} \tag{3.30}$$

$$\|\vec{n}\|_1 + (1 - r_{\text{fair}})\|\vec{n}\|_1 = (2 - r_{\text{fair}})\|\vec{n}\|_1 \tag{3.31}$$

$$\int_{p=0}^{1} p^{2\alpha_{\text{prior}} + \|\vec{k}\|_1 + r_{\text{fair}}\|\vec{n}\|_1 - 2}(1-p)^{2\beta_{\text{prior}} + -\|\vec{k}\|_1 + (2 - r_{\text{fair}})\|\vec{n}\|_1 - 2} = \frac{\Gamma(2\alpha_{\text{prior}} + \|\vec{k}\|_1 + r_{\text{fair}}\|\vec{n}\|_1 - 1)\Gamma(2\beta_{\text{prior}} - \|\vec{k}\|_1 + (2 - r_{\text{fair}})\|\vec{n}\|_1}{\Gamma(2\alpha_{\text{prior}} + 2\beta_{\text{prior}} + 2\|\vec{n}\|_1 - 2)} \tag{3.32}$$

$$= \frac{\Gamma(\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1)^2\Gamma(2\alpha_{\text{prior}} + \|\vec{k}\|_1 + r_{\text{fair}}\|\vec{n}\|_1 - 1)\Gamma(2\beta_{\text{prior}} - \|\vec{k}\|_1 + (2 - r_{\text{fair}})\|\vec{n}\|_1 - 1)}{\Gamma(\alpha_{\text{prior}} + \|\vec{k}\|_1)\Gamma(\beta_{\text{prior}} + \|\vec{n}\|_1 - \|\vec{k}\|_1)\Gamma(\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1)\Gamma(\beta_{\text{prior}} + (1 - r_{\text{fair}})\|\vec{n}\|_1)\Gamma(2\alpha_{\text{prior}} + 2\beta_{\text{prior}} + 2\|\vec{n}\|_1 - 2)} \tag{3.33}$$

Equation (3.33) isn't as tidy as we'd hoped, but at least that integral is gone. It looks very difficult to evaluate, but I have an ace up my sleeve. `mpmath` is a Python library that allows you to do calculations with arbitrary precision. This is handy when dealing with very large numbers, as conventional floating-point math breaks when the numbers get too big. One function in that library, `mpmath.gammaprod()`, allows you to calculate fractions with arbitrary numbers of Gamma functions in the numerator or denominator. It makes evaluating messes like Equation (3.33) a snap.

Let's see it in action. I'll fix $\|\vec{n}\|_1 = 28$ and $r_{\text{fair}} = \frac{1}{2}$, apply my prior of $\alpha_{\text{prior}} = \beta_{\text{prior}} = 4$, and then enumerate all possible values of $\|\vec{k}\|_1$ and plot them on a chart.

```
def posterior_H_fair( vec_k, vec_n, r_fair, a_prior, b_prior ):
    """Calculate the likelihood of H_fair, given the posterior distribution defined␣
↪by vec_n, vec_k, and the prior.
        Relies on mpmath to perform all calculations, which also means you can adjust␣
↪the precision.

    Parameters
    ----------
    vec_k: A total or list containing the successful attempts at this task until it␣
↪was completed.
    vec_n: A total or list containing the total attempts at this task until it was␣
↪completed.
    r_fair: A float between 0 and 1 representing the probability of success predicted␣
↪by H_fair.
    a_prior: A positive or zero float representing the alpha variable of the prior.
    b_prior: A positive or zero float representing the beta variable of the prior.

    Returns
    -------
    The likelihood, an mpmath float in the range [0,infinity]."""

    # place some imports here to encourage copy-paste coding
    from mpmath import gammaprod
    from numpy import sum        # vectorized, likely faster than Python's sum

    # the downside of encouraging copy-pasting is that this code will face some
    #  dirty/invalid inputs. By going wild with asserts, I'm making it tougher to
    #  use this function inappropriately.
    assert (r_fair > 0) and (r_fair < 1)
```

(continues on next page)

(continued from previous page)

```
    assert (a_prior >= 0) and (b_prior >= 0)

    # use duck typing to determine whether these are lists or not
    try:
        len_k = len(vec_k)
        k_is_list = True
    except TypeError:
        len_k = 1
        k_is_list = False

    try:
        len_n = len(vec_n)
        n_is_list = True
    except TypeError:
        len_n = 1
        n_is_list = False

    # do additional checks if both are lists
    if k_is_list and n_is_list:

        assert len_k == len_n
        for i,n in enumerate(vec_n):
            assert n > 0
            assert (vec_k[i] >= 0) and (vec_k[i] <= n)

    # now calculate sums
    if k_is_list:
        sum_k = sum(vec_k)
    else:
        sum_k = vec_k

    if n_is_list:
        sum_n = sum(vec_n)
    else:
        sum_n = vec_n

    # one final round of checks
    assert sum_n >= 1
    assert (sum_k >= 0) and (sum_k <= sum_n)

    # calculate the final result
    numerator =   [a_prior + b_prior + sum_n, a_prior + b_prior + sum_n,
                   2*a_prior + sum_k + r_fair*sum_n - 1,
                   2*b_prior - sum_k + (2 - r_fair)*sum_n - 1]
    denominator = [a_prior + sum_k, b_prior + sum_n - sum_k,
                    a_prior + r_fair*sum_n, b_prior + (1 -r_fair)*sum_n,
                   2*(a_prior + b_prior + sum_n - 1)]

    return gammaprod(numerator, denominator)
```

```
sum_n  = 28
glue( 'deffair_sum_n', sum_n, display=False )

r_fair = Fraction(1,2)
a_prior, b_prior = prior( r_fair, 4 )
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

x = np.arange(sum_n + 1)
plt.bar( x, [posterior_H_fair( v, sum_n, r_fair, a_prior, b_prior ) for v in x], \
        label=f'sum_n = {sum_n}, prior = ({a_prior}, {b_prior})')

plt.ylabel("likelihood")
plt.yticks([ 0, 2, 3.5 ])

plt.xlabel('sum_k')
plt.xticks([ 0, sum_n >> 1, sum_n ])

plt.legend()
fig_show( fig, 'fig:posterior_H_fair' )
```

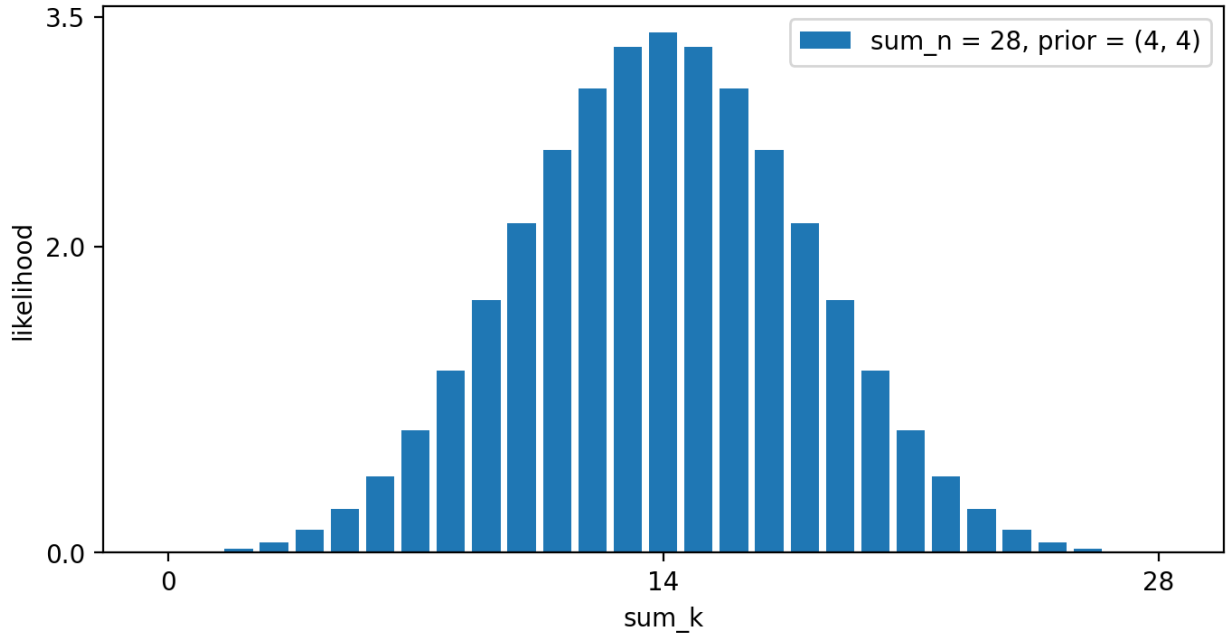

Fig. 3.6: The likelihood of all possible $\|\vec{k}\|_1$ values, when $\|\vec{n}\|_1 = 28$ and all other parameters are fixed.

This likelihood behaves much as we'd expect. $H_{\text{fair}}(\|\vec{n}\|_1)$ is maximized when $\|\vec{k}\|_1 = r_{\text{fair}}\|\vec{n}\|_1$, remains quite high for nearby values, but rapidly drops off as $\|\vec{k}\|_1$ diverges.

3.5 Defining Cheating

We're still dealing with raw likelihoods, though. As mentioned in *Missing Context*, we need a reference point. The usual solution is to calculate a Bayes factor.[KR95] You've likely seen Bayes' Theorem,

$$p(H_1|E) = \frac{p(E|H_1)p(H_1)}{p(E)} \tag{3.34}$$

The likelihood function $p(E|H_1)$ and prior $p(H_1)$ should be familiar. The normalization factor $p(E)$ is the probability of observing the evidence over all possible hypotheses, and is the canonical reference point. "All possible" is not the

same as "all," so the most common way to evaluate it is to declare a handful of hypotheses as the only ones that are possible, evaluate their likelihoods, then calculate $p(E) = p(E|H_1)p(H_1) + p(E|H_2)p(H_2) + \cdots + p(E|H_j)p(H_j)$. Defining all those hypotheses can be difficult, and opens us up to charges of cooking the books.

An alternate approach is to define just one more hypothesis and divide.

$$\frac{p(H_1|E)}{p(H_2|E)} = \frac{p(E|H_1)p(H_1)}{p(E|H_2)p(H_2)} \tag{3.35}$$

Not only is $p(E)$ eliminated, constants across the numerator and denominator cancel out. This "Bayes factor" has the same interpretation as betting odds, with numbers greater than 1 indicating H_1 is favoured and vice-versa. It has none of the flaws of p-values. For instance, since it obeys all the properties of a likelihood you can combine multiple Bayes factors together by multiplying them, just as you do with probabilities.

In this context, the only worthwhile second hypothesis is "did Dream cheat?" Unfortunately, while there's only one r_{fair} there are many ways to cheat. Ideally we'd consult with experts about Minecraft speedrunning and develop a mathematically-precise hypothesis about what cheating looks like. Since I do not have those connections, I'll invoke a backup strat.

If we accept my definition of H_{fair}, we can define H_{cheat} as the inverse of that definition; where H_{fair} asserts fair play is likely, H_{cheat} asserts it is unlikely, and vice-versa. The easiest way to accomplish that is to subtract the likelihood output by H_{fair} from some other value. This value must be high enough to prevent the result from ever being negative, and not so high that it waters down the hypothesis to look more like the Bayes/Laplace prior. The obvious choice is to use the maximal likelihood of H_{fair} as that value, which is

$$M_{\text{fair}} = \text{Beta}(\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1, \beta_{\text{prior}} + (1 - r_{\text{fair}})\|\vec{n}\|_1, m_{\text{fair}}) \tag{3.36}$$

$$m_{\text{fair}} = \frac{\alpha - 1}{\alpha + \beta - 2} = \frac{\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1 - 1}{\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1 - 2} \tag{3.37}$$

There are three flaws with using $M_{\text{fair}} - H_{\text{fair}}$ as H_{cheat}. This hypothesis gives maximal credence to both $p = 0$ and $p = 1$, and yet no cheater would dare set their probability of success to those values. Fortunately, my choice of subjective prior already factored that in. If you go with another prior, either alter it the same way or adjust $M_{\text{fair}} - H_{\text{fair}}$ appropriately.

This hypothesis also states that if $\|\vec{k}\|_1 = r_{\text{fair}}\|\vec{n}\|_1$, the likelihood of cheating is zero. Yet we can easily concoct scenarios where a run of bad luck happens to result in a record that appears perfectly fair. In comparison, H_{fair} has no zeros except at $p = 0$ and $p = 1$. A better alternative is to add some small offset to model to capture the small but ever-present possibility of cheating. It would be wise to scale this offset by the inverse of $\|\vec{n}\|_1$, to reflect the fact that as more data comes in we become more confident that cheating has not occurred.

Non-informative Priors

The scare quotes around "non-informative" are because there's a strong argument to be made all priors leak information into the posterior, as what we consider "non-informative" according to one parametrization may not be the same with another.[Syv98] It's better to think of non-informative priors as weakly informative in most circumstances, and watch carefully for the exceptions.

I propose $\frac{1}{2}\|\vec{n}\|_1^{-1}$ is a good choice. Consider an ideal Bernoulli process representing evidence towards some hypothesis. As we know nothing about this process, "non-informative" priors make more sense than a subjective ones. I consider the best-justified "non-informative" prior to be Jeffreys' prior, which in this case is Beta($\frac{1}{2}, \frac{1}{2}$). If all $\|\vec{n}\|_1$ atoms of evidence pulled from this process goes against the hypothesis, the mean value for the probability of truth p is

$$\frac{\frac{1}{2}}{\frac{1}{2} + \|\vec{n}\|_1} \approx \frac{1}{2}\frac{1}{\|\vec{n}\|_1} \tag{3.38}$$

Naturally, there's room for disagreement. If you think the Bayes/Laplace prior is less informative, then $\|\vec{n}\|_1^{-1}$ is a better offset. I can keep the constant scalar separate, so if you prefer the Bayes/Laplace you can substitute 1 in for the Jeffreys $\frac{1}{2}$, or even eliminate the offset entirely by setting the constant to zero.

Third and finally, $M_{\text{fair}} - H_{\text{fair}} + \frac{1}{2}\|\vec{n}\|_1^{-1}$ is not normalized when integrated by p, a requirement if we're to compare it against H_{fair}. This is simple to fix, as

$$\int (g(x) + h(x)) = \left(\int g(x)\right) + \left(\int h(x)\right) \tag{3.39}$$

$$\int_{p=0}^{1} t = t \cdot p|_{p=0}^{1} = t \cdot (1) - t \cdot (0) = t \tag{3.40}$$

so therefore

$$\int_{p=0}^{1} M_{\text{fair}} - \text{Beta}(\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1, \beta_{\text{prior}} + (1 - r_{\text{fair}})\|\vec{n}\|_1, p) + \frac{1}{2\|\vec{n}\|_1} = M_{\text{fair}} + \frac{1}{2\|\vec{n}\|_1} - 1 \tag{3.41}$$

and we can finally write out H_{cheat}.

$$H_{\text{cheat}}(p \mid \|\vec{n}\|_1, r_{\text{fair}}) = \frac{M_{\text{fair}} + \frac{1}{2\|\vec{n}\|_1} - \text{Beta}(\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1, \beta_{\text{prior}} + (1 - r_{\text{fair}})\|\vec{n}\|_1, p)}{M_{\text{fair}} + \frac{1}{2\|\vec{n}\|_1} - 1}, \tag{3.42}$$

$$M_{\text{fair}} = \text{Beta}(\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1, \beta_{\text{prior}} + (1 - r_{\text{fair}})\|\vec{n}\|_1, m_{\text{fair}}) \tag{3.43}$$

$$m_{\text{fair}} = \frac{\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1 - 1}{\alpha_{\text{prior}} + \beta_{\text{prior}} + \|\vec{n}\|_1 - 2} \tag{3.44}$$

For the toy example above, Equation (3.42) looks like

```
random  = np.random.default_rng(15)

k = 3      # number of Blaze rods needed
c = 5      # number of runs to earn k Blaze rods

sum_k = k * c  # we already know this
sum_n = 0

for experiment in range(c):
    results = random.choice( 2, size=16 )
    while np.sum(results) < k:
        results = np.append( results, random.choice( 2, size=16 ) )
    sum_n += np.sum( np.cumsum( results ) < k ) + 1

my_prior = prior(0.5, 4)


fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

x = np.linspace(0,1,512)
m = (my_prior[0] + sum_n/2 - 1) / (sum(my_prior) + sum_n - 2)
M = beta.pdf( m, my_prior[0] + sum_n/2, my_prior[1] + sum_n/2)
plt.fill_between( x, (M - beta.pdf( x, my_prior[0] + sum_n/2, my_prior[1] + sum_n/2)␣
↪+ (.5/sum_n))/(M-1+(.5/sum_n)), \
         label=f'H_cheat (sum_n = {sum_n})')

plt.plot( x, beta.pdf( x, my_prior[0] + sum_n/2, my_prior[1] + sum_n/2), '-k', label=f
↪'H_fair (sum_n = {sum_n})')
plt.axvline( 0.5, color='red' )
```

(continues on next page)

(continued from previous page)

```
plt.xticks([0,.5,1])
plt.xlabel('p')

plt.yticks([0,1])
plt.ylabel('likelihood')

plt.legend()
fig_show( fig, 'fig:h_cheat_h_fair_toy' )
```

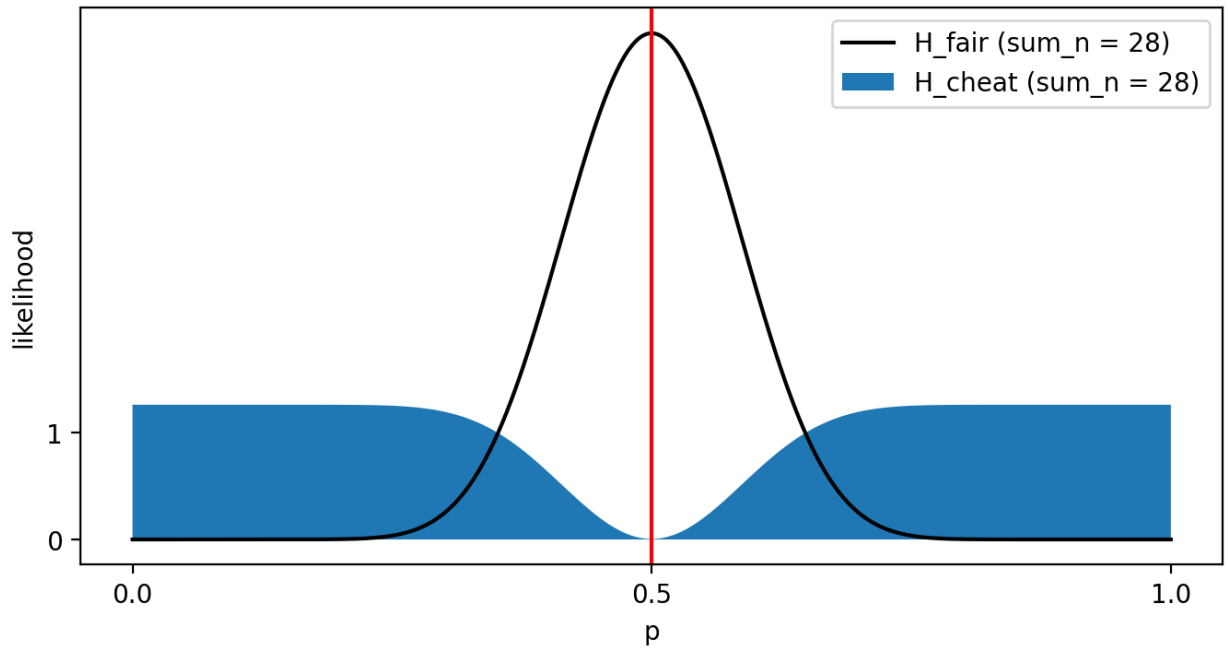

Fig. 3.7: Comparing H_{fair} with H_{cheat}, applied to the toy example.

3.6 Putting It All Together

We finally have enough math in place to calculate a Bayes factor.

$$\text{BF} = \frac{\int_{p=0}^{1} \text{Beta}(\alpha_{\text{posterior}}, \beta_{\text{posterior}}, p) H_{\text{fair}}(p \mid \|\vec{n}\|_1)}{\int_{p=0}^{1} \text{Beta}(\alpha_{\text{posterior}}, \beta_{\text{posterior}}, p) H_{\text{cheat}}(p \mid \|\vec{n}\|_1, r_{\text{fair}})} \tag{3.45}$$

$$\int_{p=0}^{1} \text{Beta}(\alpha_{\text{posterior}}, \beta_{\text{posterior}}, p) H_{\text{cheat}}(p \mid \|\vec{n}\|_1, r_{\text{fair}}) = \int_{p=0}^{1} \text{Beta}(\alpha_{\text{posterior}}, \beta_{\text{posterior}}, p) \frac{M_{\text{fair}} + \frac{1}{2\|\vec{n}\|_1} - \text{Beta}(\alpha_{\text{prior}} + r_{\text{fair}}\|\vec{n}\|_1, \beta_{\text{prior}} -}{M_{\text{fair}} + \frac{1}{2\|\vec{n}\|_1} - 1} \tag{3.46}$$

$$= \frac{1}{M_{\text{fair}} + \frac{1}{2\|\vec{n}\|_1} - 1} \left(M_{\text{fair}} + \frac{1}{2\|\vec{n}\|_1} - \int_{p=0}^{1} \text{Beta}(\alpha_{\text{posterior}}, \beta_{\text{posterior}}, p) H_{\text{fair}}(p \mid \right. \tag{3.47}$$

$$\text{BF} = \frac{\left(M_{\text{fair}} + \frac{1}{2\|\vec{n}\|_1} - 1 \right) \int_{p=0}^{1} \text{Beta}(\alpha_{\text{posterior}}, \beta_{\text{posterior}}, p) H_{\text{fair}}(p \mid \|\vec{n}\|_1)}{M_{\text{fair}} + \frac{1}{2\|\vec{n}\|_1} - \int_{p=0}^{1} \text{Beta}(\alpha_{\text{posterior}}, \beta_{\text{posterior}}, p) H_{\text{fair}}(p \mid \|\vec{n}\|_1)} \tag{3.48}$$

Notice that Equation (3.48) uses the same integral invoked to evaluate H_{fair} on the posterior, so I can recycle `posterior_H_fair()` to simplify the calculations.

```
def BF_H_fair_H_cheat( vec_k, vec_n, r_fair, a_prior, b_prior, sum_n_scalar=0.5 ):
    """Calculate the Bayes factor associated with H_fair / H_cheat, given the␣
↪posterior distribution defined by vec_n, vec_k, and the prior.
       Relies on mpmath to perform all calculations, which also means you can adjust␣
↪the precision.

    Parameters
    ----------
    vec_k: A total or list containing the successful attempts at this task until it␣
↪was completed.
    vec_n: A total or list containing the total attempts at this task until it was␣
↪completed.
    r_fair: A float between 0 and 1 representing the probability of success predicted␣
↪by H_fair.
    a_prior: A positive or zero float representing the alpha variable of the prior.
    b_prior: A positive or zero float representing the beta variable of the prior.
    sum_n_scalar: A positive or zero float that's used to weight the ||vec_n||␣
↪component. Defaults to 1/2.

    Returns
    -------
    The Bayes factor, a likelihood and mpmath float in the range [0,infinity]. Values␣
↪greater than 1 favour
       fairnes, values below 1 favour cheating."""

    from mpmath import fdiv, fmul, gammaprod, power

    # validate our one new variable
    assert sum_n_scalar >= 0

    # rely on posterior_H_fair()'s assertions to validate the remaining inputs
    integral = posterior_H_fair( vec_k, vec_n, r_fair, a_prior, b_prior )

    # but we still need sum_n for calculations
    try:
        len_n = len(vec_n)
        n_is_list = True
    except TypeError:
        len_n = 1
        n_is_list = False

    if n_is_list:
        sum_n = sum(vec_n)
    else:
        sum_n = vec_n


    # invoke mpmath instead of python's functions, to discourage precision loss
    m = fdiv( a_prior + r_fair*sum_n - 1,  a_prior + b_prior + sum_n - 2 )
    M = fmul( gammaprod([a_prior + b_prior + sum_n], [a_prior + r_fair*sum_n,b_prior␣
↪+ (1-r_fair)*sum_n]), \
             fmul(power( m, a_prior + r_fair*sum_n - 1 ), power( 1-m, b_prior + (1-r_
↪fair)*sum_n - 1 )) )

    return fdiv( fmul(M - 1 + sum_n_scalar/sum_n, integral), M + sum_n_scalar/sum_n -␣
↪integral )
```

(continues on next page)

(continued from previous page)

Let's see `BF_H_fair_H_cheat()` in action.

```
sum_n  = 28
r_fair = Fraction(1,2)
a_prior, b_prior = prior( r_fair, 4 )


fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

x = np.arange(sum_n + 1)
plt.plot( x, [BF_H_fair_H_cheat( v, sum_n, float(r_fair), a_prior, b_prior ) for v in␣
↪x], '-', \
        label=f'sum_n = {sum_n}, prior = ({a_prior}, {b_prior})')
plt.plot( x, [1 for v in x], '-', \
        label='break even')

plt.ylabel("H_fair / H_cheat")
plt.yticks([0,1,2,4,8])

plt.xlabel('sum_k')
plt.xticks([ 0, 8, 14, 20, 28 ])

plt.legend()
fig_show( fig, 'fig:BF_posterior_H_fair' )
```

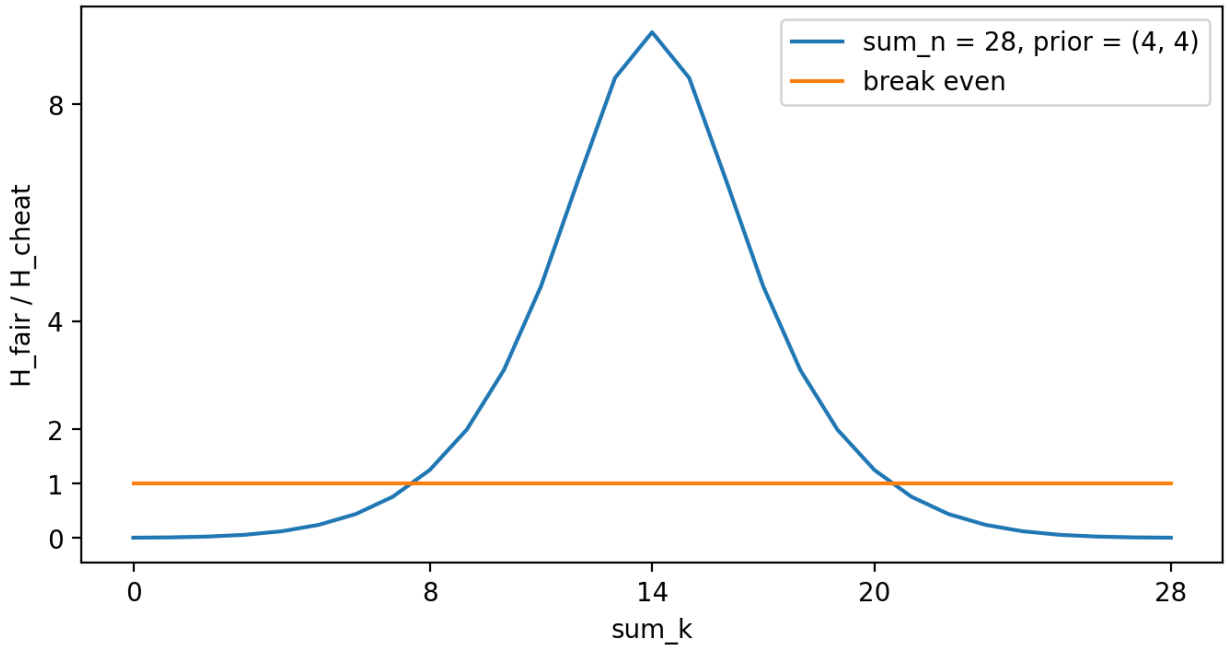

Fig. 3.8: The Bayes factor for all possible $\|\vec{k}\|_1$ values, when $\|\vec{n}\|_1 = 28$ and all other parameters are fixed.

The results are as expected. The sharper peak relative to *the other chart* with H_{fair} alone is due to the increasing likelihood of H_{cheat} given a $\|\vec{k}\|_1$ that diverges from perfect fairness. Values of $\|\vec{k}\|_1$ near r_{fair} are nonetheless considered net evidence in favour of H_{fair}, while values far from there count as better evidence for H_{cheat}. The crossover point is about six and a half possible values of $\|\vec{k}\|_1$ away from perfect fairness in this case, though that point should be

different for different $\|\vec{n}\|_1$.

Another useful test is to fix $\|\vec{k}\|_1 = \frac{1}{2}\|\vec{n}\|_1$ and increase $\|\vec{n}\|_1$. If you've done much frequentist analysis, you've noticed the term $\sqrt{n}$ seems to pop up everywhere. This is a side-effect of the central limit theorem applied to "well-behaved" distributions, where it's been proven that estimates of true mean will converge to that value at a rate proportional to $\sqrt{n}$. We can think of that value as a metric for how strong the evidence is. Since Bernoulli processes are "well-behaved," our evidence for any fixed hypothesis should show the same scaling factor.

```
r_fair = 0.5
a_prior, b_prior = prior( r_fair, 4 )


fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

x = np.exp( np.linspace(np.log(8), np.log(1 << 18), 256) )
plt.plot( x, [BF_H_fair_H_cheat( v/2, v, float(r_fair), a_prior, b_prior ) for v in␣
↪x], '-', \
        label=f'maximal likelihood')

plt.plot( x, 1.9*np.sqrt(x), '-k', label='1.9 * sqrt(sum_n)' )

plt.ylabel("H_fair / H_cheat")
plt.yticks([1, 1000])

plt.xlabel('sum_n')
plt.xticks([8, 1 << 18])

plt.legend()
fig_show( fig, 'fig:weight_evidence' )
```

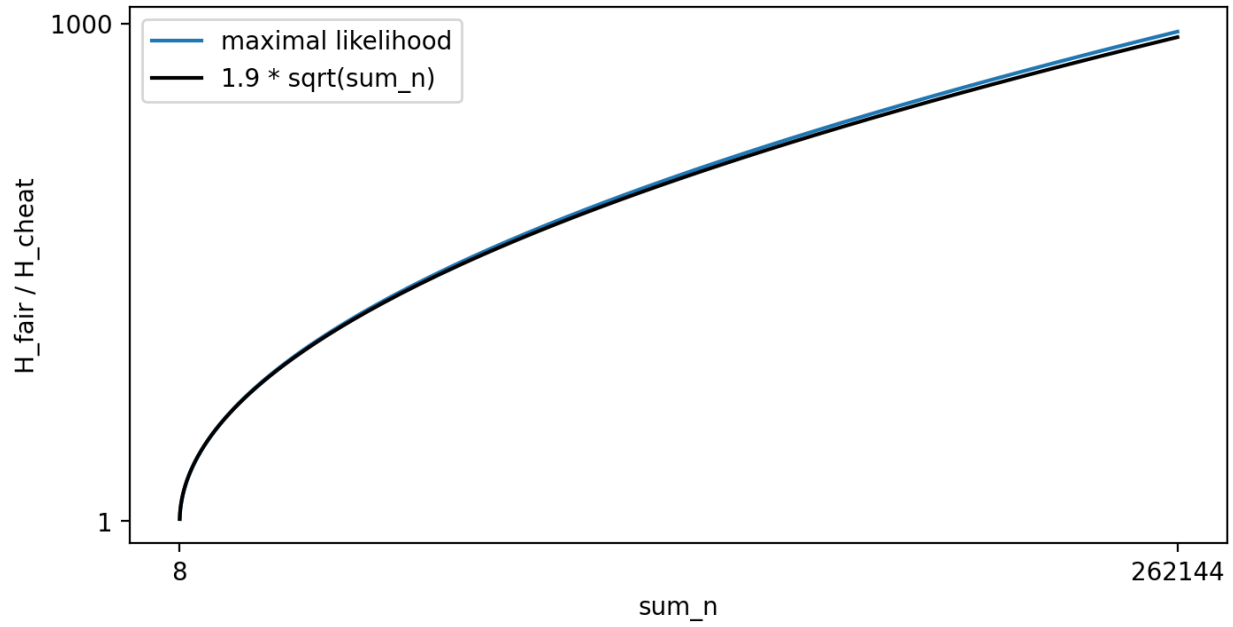

Fig. 3.9: How the Bayes factor increases when $\|\vec{k}\|_1 = \frac{1}{2}\|\vec{n}\|_1$.

Again, *this chart* holds no surprises, save a subtle divergence between the theoretical and practical behaviour of the Bayes factor. This is likely due to the $\frac{1}{2}\|\vec{n}\|_1^{-1}$ term adding some skew when $\|\vec{n}\|_1$ is small.

CHAPTER

FOUR

SIMULATION RESULTS

As nice as all that mathematics was, it means nothing if this Bayes factor doesn't work as advertised. If it doesn't increase when a person plays with an unaltered copy of the game, and if it doesn't decrease when a cheater modifies their drop or barter rate, then it is useless no matter what the math tells us.

Modifying a copy of Minecraft 1.16.1 and doing speed runs is a time-consuming process, but we have a viable alternative: since the decision of whether to drop a Blaze rod or collection of Ender pearls is a Bernoulli process, it can be perfectly simulated by any other Bernoulli process. Pseudo-random number generators are a particularly good choice for this, as the probability of success can be controlled directly. We must still take care to match the behaviour of Minecraft speedrunners as closely as we can, of course.

```
# Execute this cell to install all dependencies. Apologies for the spam.

# If you're running this on your own computer, I recommend altering "install mpmath"
#  to read "install --user mpmath", that way you don't need administrator access.
# You can get rid of the spam by changing "pip install" to "pip -q install", but on␣
↪Colab
#  you could miss a message that you need to restart your runtime. If you don't␣
↪restart,
#  the code won't work, hence why the default is to spam.
# A few error messages are fine, I usually get one about mismatched versions and the␣
↪code
#  still runs.

!pip install mpmath myst_nb numpy pandas matplotlib scipy
```

```
Requirement already satisfied: mpmath in /usr/lib/python3/dist-packages (1.1.0)
Requirement already satisfied: myst_nb in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (0.10.2)
```

```
Requirement already satisfied: numpy in /usr/lib/python3/dist-packages (1.18.4)
Requirement already satisfied: pandas in /usr/lib/python3/dist-packages (1.0.5)
Requirement already satisfied: matplotlib in /usr/lib/python3/dist-packages (3.3.0)
Requirement already satisfied: scipy in /usr/lib/python3/dist-packages (1.5.2)
Requirement already satisfied: importlib-metadata in /usr/lib/python3/dist-packages␣
↪(from myst_nb) (1.6.0)
```

```
Requirement already satisfied: myst-parser~=0.12.9 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (0.12.10)
```

```
Requirement already satisfied: pyyaml in /usr/lib/python3/dist-packages (from myst_
↪nb) (5.3.1)
Requirement already satisfied: jupyter-cache~=0.4.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (0.4.1)
```

```
Requirement already satisfied: nbformat~=5.0 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from myst_nb) (5.0.8)
Requirement already satisfied: ipython in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from myst_nb) (7.19.0)
```

```
Requirement already satisfied: sphinx-togglebutton~=0.2.2 in /home/hjhornbeck/.local/
↪lib/python3.8/site-packages (from myst_nb) (0.2.3)
```

```
Requirement already satisfied: nbconvert~=5.6 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from myst_nb) (5.6.1)
```

```
Requirement already satisfied: jupyter-sphinx==0.3.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (0.3.1)
Requirement already satisfied: ipywidgets<8,>=7.0.0 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (7.6.3)
```

```
Requirement already satisfied: sphinx<4,>=2 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from myst_nb) (3.4.3)
```

```
Requirement already satisfied: docutils>=0.15 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from myst_nb) (0.16)
Requirement already satisfied: markdown-it-py~=0.5.4 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst-parser~=0.12.9->myst_nb) (0.5.8)
```

```
Requirement already satisfied: nbclient<0.6,>=0.2 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from jupyter-cache~=0.4.1->myst_nb) (0.5.1)
```

```
Requirement already satisfied: nbdime in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from jupyter-cache~=0.4.1->myst_nb) (2.1.0)
```

```
Requirement already satisfied: sqlalchemy~=1.3.12 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from jupyter-cache~=0.4.1->myst_nb) (1.3.22)
```

```
Requirement already satisfied: attrs in /usr/lib/python3/dist-packages (from jupyter-
↪cache~=0.4.1->myst_nb) (19.3.0)
```

```
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/lib/python3/dist-
↪packages (from nbformat~=5.0->myst_nb) (3.2.0)
Requirement already satisfied: ipython-genutils in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from nbformat~=5.0->myst_nb) (0.2.0)
```

```
Requirement already satisfied: traitlets>=4.1 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from nbformat~=5.0->myst_nb) (5.0.5)
Requirement already satisfied: jupyter-core in /usr/lib/python3/dist-packages (from␣
↪nbformat~=5.0->myst_nb) (4.6.3)
Requirement already satisfied: jedi>=0.10 in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (0.17.0)
Requirement already satisfied: decorator in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (4.4.2)
Requirement already satisfied: backcall in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (0.2.0)
Requirement already satisfied: pygments in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from ipython->myst_nb) (2.7.3)
```

```
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/
↪lib/python3/dist-packages (from ipython->myst_nb) (3.0.6)
Requirement already satisfied: setuptools>=18.5 in /usr/lib/python3/dist-packages␣
↪(from ipython->myst_nb) (49.3.1)
Requirement already satisfied: pexpect>4.3; sys_platform != "win32" in /usr/lib/
↪python3/dist-packages (from ipython->myst_nb) (4.6.0)
Requirement already satisfied: pickleshare in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (0.7.5)
Requirement already satisfied: wheel in /usr/lib/python3/dist-packages (from sphinx-
↪togglebutton~=0.2.2->myst_nb) (0.34.2)
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/lib/python3/dist-packages␣
↪(from nbconvert~=5.6->myst_nb) (0.8.4)
```

```
Requirement already satisfied: testpath in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (0.4.4)
Requirement already satisfied: defusedxml in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (0.6.0)
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/lib/python3/dist-packages␣
↪(from nbconvert~=5.6->myst_nb) (1.4.2)
Requirement already satisfied: entrypoints>=0.2.2 in /usr/lib/python3/dist-packages␣
↪(from nbconvert~=5.6->myst_nb) (0.3)
Requirement already satisfied: jinja2>=2.4 in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (2.11.2)
Requirement already satisfied: bleach in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (3.2.1)
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >= "3.6" in /
↪home/hjhornbeck/.local/lib/python3.8/site-packages (from ipywidgets<8,>=7.0.0->myst_
↪nb) (1.0.0)
```

```
Requirement already satisfied: widgetsnbextension~=3.5.0 in /home/hjhornbeck/.local/
↪lib/python3.8/site-packages (from ipywidgets<8,>=7.0.0->myst_nb) (3.5.1)
Requirement already satisfied: ipykernel>=4.5.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from ipywidgets<8,>=7.0.0->myst_nb) (5.4.2)
Requirement already satisfied: sphinxcontrib-devhelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.2)
```

```
Requirement already satisfied: sphinxcontrib-applehelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.2)
Requirement already satisfied: sphinxcontrib-serializinghtml in /home/hjhornbeck/.
↪local/lib/python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.1.4)
```

```
Requirement already satisfied: sphinxcontrib-htmlhelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.3)
Requirement already satisfied: sphinxcontrib-qthelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.3)
Requirement already satisfied: snowballstemmer>=1.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (2.0.0)
Requirement already satisfied: requests>=2.5.0 in /usr/lib/python3/dist-packages␣
↪(from sphinx<4,>=2->myst_nb) (2.23.0)
```

```
Requirement already satisfied: sphinxcontrib-jsmath in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.1)
Requirement already satisfied: alabaster<0.8,>=0.7 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (0.7.12)
Requirement already satisfied: imagesize in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from sphinx<4,>=2->myst_nb) (1.2.0)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: packaging in /usr/lib/python3/dist-packages (from␣
↪sphinx<4,>=2->myst_nb) (20.4)
```

```
Requirement already satisfied: babel>=1.3 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from sphinx<4,>=2->myst_nb) (2.9.0)
Requirement already satisfied: nest-asyncio in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from nbclient<0.6,>=0.2->jupyter-cache~=0.4.1->myst_nb) (1.4.3)
Requirement already satisfied: async-generator in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from nbclient<0.6,>=0.2->jupyter-cache~=0.4.1->myst_nb) (1.10)
Requirement already satisfied: jupyter-client>=6.1.5 in /usr/lib/python3/dist-
↪packages (from nbclient<0.6,>=0.2->jupyter-cache~=0.4.1->myst_nb) (6.1.6)
Requirement already satisfied: six in /usr/lib/python3/dist-packages (from nbdime->
↪jupyter-cache~=0.4.1->myst_nb) (1.15.0)
Requirement already satisfied: tornado in /usr/lib/python3/dist-packages (from nbdime-
↪>jupyter-cache~=0.4.1->myst_nb) (6.0.4)
Requirement already satisfied: notebook in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from nbdime->jupyter-cache~=0.4.1->myst_nb) (6.1.6)
```

```
Requirement already satisfied: GitPython!=2.1.4,!=2.1.5,!=2.1.6 in /home/hjhornbeck/.
↪local/lib/python3.8/site-packages (from nbdime->jupyter-cache~=0.4.1->myst_nb) (3.1.
↪12)
Requirement already satisfied: colorama in /usr/lib/python3/dist-packages (from␣
↪nbdime->jupyter-cache~=0.4.1->myst_nb) (0.4.3)
```

```
Requirement already satisfied: pytz>=2015.7 in /usr/lib/python3/dist-packages (from␣
↪babel>=1.3->sphinx<4,>=2->myst_nb) (2020.1)
Requirement already satisfied: pyzmq>=17 in /usr/lib/python3/dist-packages (from␣
↪notebook->nbdime->jupyter-cache~=0.4.1->myst_nb) (19.0.2)
Requirement already satisfied: Send2Trash in /usr/lib/python3/dist-packages (from␣
↪notebook->nbdime->jupyter-cache~=0.4.1->myst_nb) (1.5.0)
Requirement already satisfied: terminado>=0.8.3 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from notebook->nbdime->jupyter-cache~=0.4.1->myst_nb) (0.9.
↪2)
Requirement already satisfied: prometheus-client in /usr/lib/python3/dist-packages␣
↪(from notebook->nbdime->jupyter-cache~=0.4.1->myst_nb) (0.7.1)
Requirement already satisfied: argon2-cffi in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from notebook->nbdime->jupyter-cache~=0.4.1->myst_nb) (20.1.0)
```

```
Requirement already satisfied: gitdb<5,>=4.0.1 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from GitPython!=2.1.4,!=2.1.5,!=2.1.6->nbdime->jupyter-cache~=0.4.
↪1->myst_nb) (4.0.5)
```

```
Requirement already satisfied: ptyprocess; os_name != "nt" in /home/hjhornbeck/.local/
↪lib/python3.8/site-packages (from terminado>=0.8.3->notebook->nbdime->jupyter-cache~
↪=0.4.1->myst_nb) (0.7.0)
Requirement already satisfied: cffi>=1.0.0 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from argon2-cffi->notebook->nbdime->jupyter-cache~=0.4.1->myst_nb)␣
↪(1.14.4)
Requirement already satisfied: smmap<4,>=3.0.1 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from gitdb<5,>=4.0.1->GitPython!=2.1.4,!=2.1.5,!=2.1.6->nbdime->
↪jupyter-cache~=0.4.1->myst_nb) (3.0.4)
Requirement already satisfied: pycparser in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from cffi>=1.0.0->argon2-cffi->notebook->nbdime->jupyter-cache~=0.4.
↪1->myst_nb) (2.20)
```

```
from fractions import Fraction

from math import log,factorial
import matplotlib.pyplot as plt
from mpmath import mp
from myst_nb import glue

import numpy as np

import pandas as pd
from scipy.optimize import differential_evolution
from scipy.stats import beta,binom,nbinom


dpi         = 200  # change this to increase/decrease the resolution charts are made␣
↪at
book_output = True # changing this to False will make it easier to view plots in a␣
↪Jupyter notebook

def fig_show( fig, name ):
    """A helper to control how we're displaying figures."""
    global book_output

    if book_output:
        glue( name, fig, display=False )
        plt.close();
    else:
        fig.show()
```

```
# missing "simple_bernoulli_bf.py"? Uncomment this line to grab it from the repository
# !wget -c "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_cheating/main/
↪simple_bernoulli_bf.py"

from simple_bernoulli_bf import prior, posterior_H_fair, BF_H_fair_H_cheat
```

4.1 Blaze Rod Drops

Blaze rod drops are the easiest to model, as it almost certainly follows the Negative Binomial distribution. To try and match what actual speedrunners do, we'll load up all the data from the MST report and pool it together.

```
# missing the data files? Uncommenting and running this cell might retrieve them

# !mkdir data
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.benex.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.dream.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.illumina.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.sizzler.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.vadikus007.tsv"
```

```
# create a blank dataframe to store all this data in
all_blaze_rods = pd.DataFrame( {'n':[], 'k':[]} )


# the Ender pearl stats have slightly different names
blaze_players = ['benex','dream','illumina','sizzler','vadikus007']

# load in the data
for p in blaze_players:
    all_blaze_rods = all_blaze_rods.append( pd.read_csv(f'data/blaze.{p}.tsv',sep="\t
↪") )


fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')


# pull out all the unique values to be the X axis along our bar chart
blaze_k = sorted( all_blaze_rods['k'].unique() )
plt.bar( blaze_k, [all_blaze_rods['k'][ all_blaze_rods['k'] == k ].count() for k in␣
↪blaze_k], \
        label='blaze rods gathered' )

plt.xlabel('blaze rods gathered')
plt.ylabel('number of runs where k rods were gathered')
plt.yticks([0,15,30])

fig_show( fig, 'fig:blaze_rods_gathered' )
```

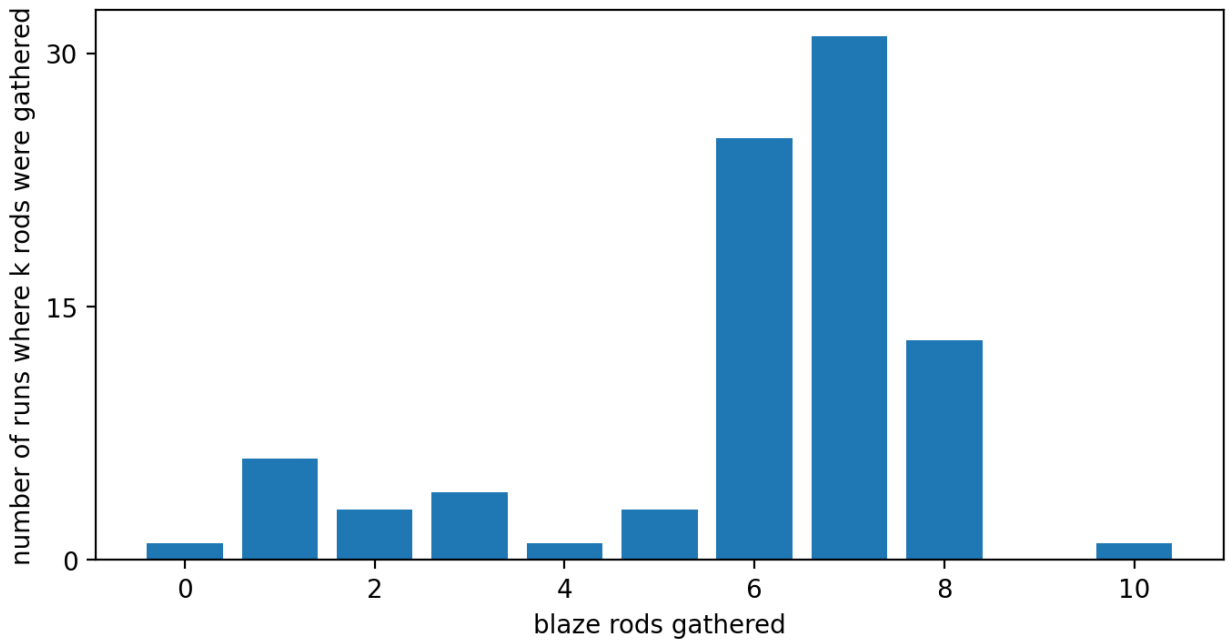

Fig. 4.1: The number of Blaze rods gathered during runs from five separate top-level speedrunners, according to the Minecraft Speeedrunning Team.

4.1.1 The Mechanics of Blaze Rods

There's a surprising amount of variance in *the number of Blaze rods gathered*, given the game mechanics involved.

Accessing The End requires finding a Stronghold, and finding that requires tossing Eyes of Ender to point out the direction of the Stronghold. Triangulating the location requires a minimum of two Eyes, and there's a one in five chance an Eye will break when tossed. Once found, players must find a portal within the Stronghold that grants them access to The End. This portal is powered by twelve Eyes of Ender, which have a one-in-ten chance of spontaneously appearing in an appropriate spot. Do the math, and the most common outcome is to have one Eye of Ender already in the portal, while three or more Eyes are present 34.1% of the time.

Eyes of Ender are crafted by combining Ender pearls with Blaze powder. Two items of the latter can be created from one Blaze rod. Given all of the above, the safest route is to kill Blazes until the runner has seven Blaze rods, which allows for fourteen Blaze powders and thus fourteen Eyes of Ender. Even if two Eyes of Ender break when finding the Stronghold, which has a 4% chance of happening, the runner has enough Eyes to fill an empty portal.

While the safest route is indeed the most common one, a substantial number of speedruns collected only six rods. This is a viable route, as immediate entry to The End either requires no Eyes break while finding the stronghold (64% odds), only one Eye breaks but the portal has at least one Eye already present (about 23% odds), or both eyes break but two or more Eyes are present (about 1.4%). Surprisingly, a number of runners also collected eight rods. This may be because the runner wasn't confident they could find the Stronghold with just two Eyes and wanted some safety margin.

Five rods is the bare minimum I could see used to complete a speed run, as the odds of a runner with only eight Eyes of Ender keeping both the ones they toss and finding four or more in the End portal are only 1.6%. The low-rod runs were likely aborted due to other reasons. That one run of ten rods comes from Benex, and the comments about going after Wither skeletons suggest it wasn't a typical speedrun.

In real life, speed runners pick a range of Blaze rod targets, so our simulation should do the same. We'll discard all the recorded runs where less than five and more than eight rods were collected, and use their relative frequencies to pick 200 values for k. We'll then create three universes: one where players with an unmodified game have $\vec{k}$ for Blaze rod targets, one where players boost their drop odds for the same targets, and one where players reduce their odds. Since one random simulation doesn't give you a feel for the variation present, we'll run 16 distinct players for each of the three scenarios just mentioned.

```
random  = np.random.default_rng(42)

# generate the weights and normalize
weights = np.array( [all_blaze_rods['k'][ all_blaze_rods['k'] == k ].count() for k in␣
↪range(5,9)] )
weights = weights / np.sum(weights)

r_fair    = 0.5
r_cheat_h = 0.569 ; glue( 'sim_blaze_r_cheat_h', r_cheat_h*100, display=False )
r_cheat_l = 0.431 ; glue( 'sim_blaze_r_cheat_l', r_cheat_l*100, display=False )
count     = 200   ; glue( 'sim_blaze_count', count, display=False )
clones    = 16    ; glue( 'sim_blaze_clones', clones, display=False )
a_prior, b_prior   = prior(r_fair, 4)

x      = np.arange(1, count+1)
k      = 5 + random.choice( 4, size=(clones,count), p=weights )
sum_k  = np.cumsum( k, axis=1 )

cheat_h_nk = random.negative_binomial( k, r_cheat_h )
cheat_l_nk = random.negative_binomial( k, r_cheat_l )
fair_nk    = random.negative_binomial( k, r_fair )

sum_k         = np.cumsum( k, axis=1 )
```

(continues on next page)

(continued from previous page)

```
sum_cheat_h_n = sum_k + np.cumsum( cheat_h_nk, axis=1 )
sum_cheat_l_n = sum_k + np.cumsum( cheat_l_nk, axis=1 )
sum_fair_n    = sum_k + np.cumsum( fair_nk, axis=1 )


fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

for idx in range(clones):
    y = [BF_H_fair_H_cheat(sum_k[idx][i], sum_cheat_h_n[idx][i], r_fair, a_prior, b_
↪prior) for i in range(count)]
    if idx == 0:
        plt.plot( x, y, '-r', label=f'cheater (r={r_cheat_h:.3f})' )
    else:
        plt.plot( x, y, '-r', alpha=0.1 )

    y = [BF_H_fair_H_cheat(sum_k[idx][i], sum_cheat_l_n[idx][i], r_fair, a_prior, b_
↪prior) for i in range(count)]
    if idx == 0:
        plt.plot( x, y, '-b', label=f'cheater (r={r_cheat_l:.3f})' )
    else:
        plt.plot( x, y, '-b', alpha=0.1 )

    y = [BF_H_fair_H_cheat(sum_k[idx][i], sum_fair_n[idx][i], r_fair, a_prior, b_
↪prior) for i in range(count)]
    if idx == 0:
        plt.plot( x, y, '-k', label=f'fair (r={r_fair:.3f})' )
    else:
        plt.plot( x, y, '-k', alpha=0.1 )

plt.plot( x, [1 for v in x], '--g', label='break even' )

plt.xlabel("rounds")
plt.xticks([1,count])
plt.ylabel('H_fair / H_cheat')
plt.yscale("log")

plt.legend()
fig_show( fig, 'fig:sim_blaze_trio' )
```

There are *two distinct behaviours*. For players with unmodified drop rates, the Bayes factor usually stays above one and appears to shadow the $\sqrt{n}$ trajectory detailed earlier. For the two cheating players, the Bayes factor gradually drifts below one and in some cases gets below 1:1,000,000 odds of playing fair. There's a bit of variation, though, as one of the fair players' Bayes factors drifts downward toward the 1:1 odds range, and one of the cheaters manages roughly 1:1 odds of fair play during the entire session. There are also times where the cheating players have Bayes factors that suggest fair play, those these usually only happen during the first rounds.

Interestingly, the cheater that reduced their odds of a Blaze rod drop seems to have fared worse than the player that boosted their Blaze rod drops. One explanation is that while the Negative Binomial may be treated the same as the Binomial in the long run, always stopping on a positive will skew the results in the short term.

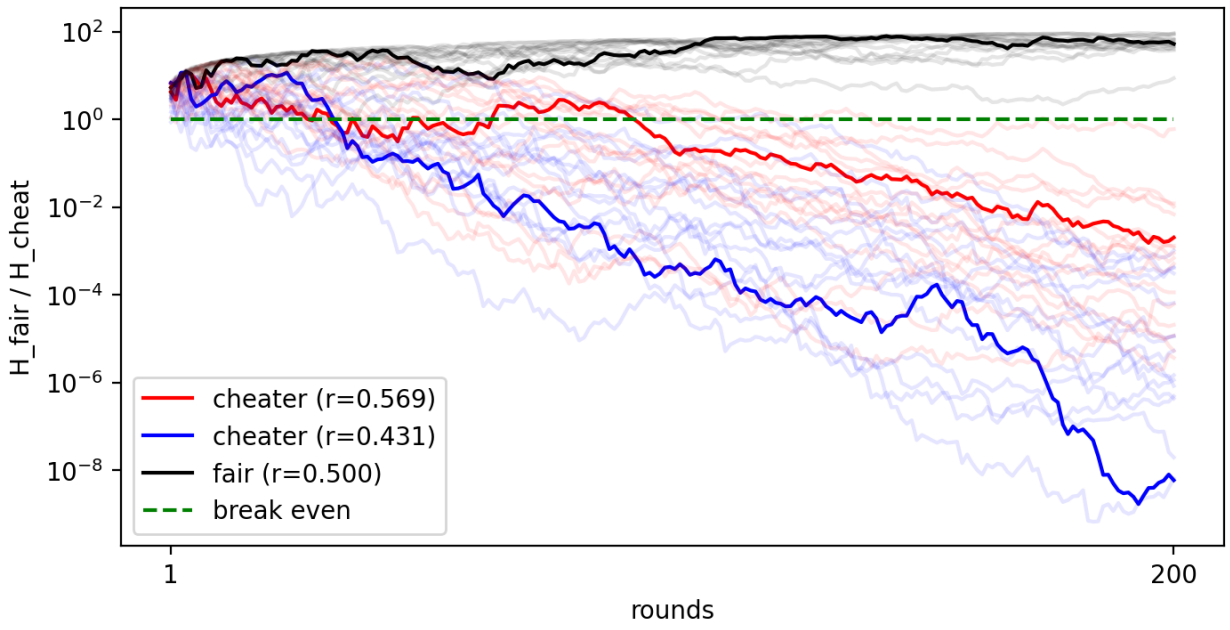

Fig. 4.2: The behaviour of this Bayes Factor for 16 clones of three players that had different Blaze rod drop rates. One group of 16 clones had the usual drop rate, one group increased it to 56.9%, and one decreased it to 43.1%

4.2 Cheating Techniques

Why would a cheater want to reduce the number of Blaze rods they collect? Put yourself behind their keyboard. They know that the odds of being detected are high if they simply increase their drop rate. Someone watching the stream will get suspicious, count up Blaze rods, and alert others.

A big-brained alternative is to *temporarily* elevate the drop rate. We're used to thinking of people getting lucky from time to time, so a short run of unusually good luck might pass under the radar. Even if it doesn't, the cheater has a plausible excuse. They can train during the sessions with the fair drop rate, and only start cheating when they're skilled enough to earn a record.

One problem: for them, it will always be true that $\|\vec{k}\|_1 > r_{\text{fair}}\|\vec{n}\|_1$. No one may be able to detect their cheating in the short term, but a career-long analysis will show an elevated success rate relative to non-cheating players. The more times this cheater temporarily boosts their luck, the greater the evidence they've done exactly that.

A galaxy-brained move is to alternate between short periods of elevated drop rates and long periods of *deflated* drop rates. The PE report considers multiple "boost rates" for Ender pearl barters, but never considers deflated rates. The MST report states that apparently elevated drop rates made the speedrunning community suspicious of Dream (pg. 3), and that any runner "observed experiencing such unlikely events would be held to the same level of scrutiny" (pg. 4); the flip side is that any runner experiencing an unusually low number of drops would not. This blind spot of the speedrunning community could be exploited: by altering their drop rate both up and down, a cheating speedrunner can ensure their career stats do not reveal evidence of cheating.

Greatly reducing their drop rate creates a large handicap to overcome, even if it does mean they can punch the "cheat" button pretty often. Instead, long periods of slightly-decreased luck would reduce the handicap and allow the cheater to exploit unusually good luck during their grind times. Should one of their elevated periods raise suspicion, not only can they pull out the "I got lucky" excuse, they can also point to a specific period of prior attempts and argue that, when taken as a whole, all the evidence demonstrates their drop rate is normal. The PE report analyzed the six attempts the MST report did for evidence of elevated drop rates, as well as those six plus five attempts Dream catalogued, but never considered analyzing those five attempts in isolation.

One or Two tails?

P-values are Not What You Think mentioned that p-values have a problem with the definition of "extreme." Say I assert that a coin is biased by being more likely to come up heads than tails. I toss the coin, and get far more tails than heads. Depending on where you place the emphasis, I've either proven my assertion to be correct or incorrect. Worse, since the line for "extreme" is further from the centre with a two-tailed test verses a one-tailed test, researchers have a bit of "tilt" to shift values near that line to the side of their choosing. Worse still, there are no unambiguous standards for which type of p-value to use.[Pil91]

To be clear, I'm not accusing Dream of either type of cheating. I've deliberately avoided doing the appropriate analysis before writing this text, in fact. When analyzing both reports for flaws, though, this oversight is easy to spot. It is also easy to detect, once you know what to look for, and the Bayes factor I've created is sensitive to both increased and decreased drop rates. An analysis that uses p-values might also be able to detect this if a "two-tailed" p-value is used, but the MST report opted to use a "one-tailed" p-value instead.

4.3 Ender Pearl Barters

This analysis has focused on Blaze rods drops because they are so easy to understand. They can be directly analogized to the flip of a coin, and quite clearly follow a Negative Binomial, so the choice of simulation is very clear. None of the above math assumed a fifty-fifty success rate, nor whether the numbers were better modelled by the Negative Binomial or Binomial distributions. It should work flawlessly with Ender pearl barters.

Our simulation, on the other hand, must choose between the Negative Binomial or Binomial.

```
# missing the data files? Uncommenting and running this cell might retrieve them

# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.benex.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.dream_before.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.dream_after.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.illumina.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.sizzler.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.vadikus007.tsv"
```

```
# create a blank dataframe to ensure the proper format
all_pearl_barters = pd.DataFrame( {'n':[], 'k':[]} )


# the Blaze rod stats have slightly different names
pearl_players = ['benex','dream_before','dream_after','illumina','sizzler','vadikus007
↪']

# load in the data
for p in pearl_players:
    all_pearl_barters = all_pearl_barters.append( pd.read_csv(f'data/bartering.{p}.tsv
↪',sep="\t") )
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

# pull out all the unique values for the X axis
pearl_k = sorted( all_pearl_barters['k'].unique() )
plt.bar( pearl_k, [all_pearl_barters['k'][ all_pearl_barters['k'] == k ].count() for␣
↪k in pearl_k], \
        label='ender pearl barters' )

plt.xlabel('successful barters')
plt.ylabel('number of runs where k barters were successful')
plt.yticks([0,25,50])

fig_show( fig, 'fig:ender_pearl_barter' )
```

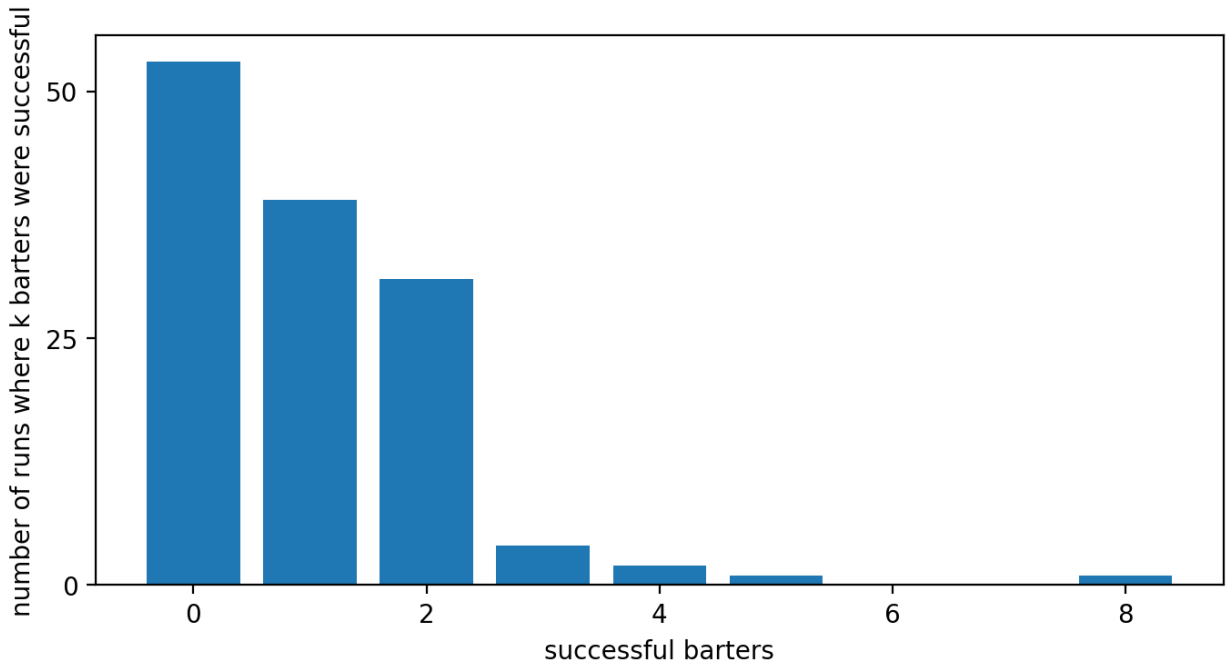

Fig. 4.3: The number of Ender pearl barters performed during a speedrun, according to the Minecraft Speeedrunning Team.

In *Overuse of the Negative Binomial*, I gave a theoretical argument for why Ender pearl barters may not follow a Negative Binomial distribution. Now we have *an empirical argument*. Bartering with Piglins is supposed to be the primary way to earn Ender pearls in a random-seed any-percent Minecraft 1.16 speedrun, and yet most speed runners earned zero Ender pearls via this method?

The easiest explanation is that few of the runs in the MST report were successful. Players usually target a specific number of Ender pearls, but barters drop anywhere from four to eight pearls. Assuming uniform odds, two Ender pearl barters have a 60% chance of giving twelve or more pearls, and as long as none of the resulting Eyes of Ender break while finding the Stronghold the runner can immediately go to The End. Half the time, it takes anywhere from two to thirty-five barters for a speed runner to earn two Ender pearl barters. Conversely, this means that half the time a player will have to wait longer than three and a half minutes to earn those barters, a serious problem if you are on the clock. If we assume that Minecraft 1.16 runners always target two barters, then only a quarter of all speed runs are successful. If players only stop bartering if they are out of time or earn those two barters, then that success rate implies they budget for twenty barters, which requires exactly two minutes to complete, and abort if they fail to hit that target.

Of course, that assumes quite a bit. Endermen have a fifty-fifty chance of dropping an Ender pearl when killed. While they rarely spawn in most Minecraft biomes, one notable exception is Nether Warped forest. A player that happens to stumble on such a biome can use that instead to gather Ender pearls. Barters also require gold ingots, so it is also possible for a speed runner to abort if they run out of gold instead of time. Theoretically, it's also possible to get to The End with just one Ender pearl barter. There's a 20% chance of getting eight Ender pearls at once and a 50% chance of picking up two more via killing two to three Enderman, which gives the runner a 25% chance of jumping immediately to The End once they've found the Stronghold. The overall odds of that happening are about 2.5%, but a speedrunner may prefer the extra practice instead of aborting immediately.

Nonetheless, this gives us a prediction: if time or gold is more likely to be the stopping criteria than the number of successful Ender pearl barters, and both are correlated to n rather than k, then charting the data with n on the X axis should reveal a cutoff value for n.

```
r_pearl = 20/423

# pull out all the unique values
pearl_n = all_pearl_barters['n'].unique()
x       = np.arange(int(max( pearl_n )))

# separate these so we can scale the negative binomial appropriately
pearl_y   = [all_pearl_barters['n'][ \
                                    (all_pearl_barters['n'] == n) & (all_pearl_
↪barters['k'] < 3) ].count() \
             for n in pearl_n]
nbinom2_y = nbinom( 2, r_pearl ).pmf( x )
nbinom2_y = nbinom2_y / np.max(nbinom2_y) * np.max(pearl_y)

fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

plt.bar( pearl_n, pearl_y, label='Ender pearl barters' )
plt.plot( x + 2, nbinom2_y, '-k', label='Negative Binomial, k=2', alpha=0.3 )

plt.xlabel('number of barters')
plt.xticks([1,10,15,20,25,100])
plt.ylabel('number of runs with n barters')
plt.yticks([0,3,9])

plt.legend()
fig_show( fig, 'fig:barters' )
```

The data does show a bit of a cliff at the $n = 20$ mark, but it's more of a mesa with tonnes of debris at the base. There isn't a hard stopping time, it looks more like players trail off after that time. There's also an unusual number of runs that end around $n = 10$, much more than we'd expect from a Negative Binomial, and an unusual lack of runs in the teens. This might be due to where players decide to stop.

```
r_pearl = 20/423

x       = np.arange(1,int(all_pearl_barters['n'].max()))

fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')
axs = fig.subplots(3, 1)

for k in range(3):

    # we only care about a specific k
    mask = all_pearl_barters['k'] == k
    pearl_x = all_pearl_barters[ mask ]['n'].unique()
```

(continues on next page)

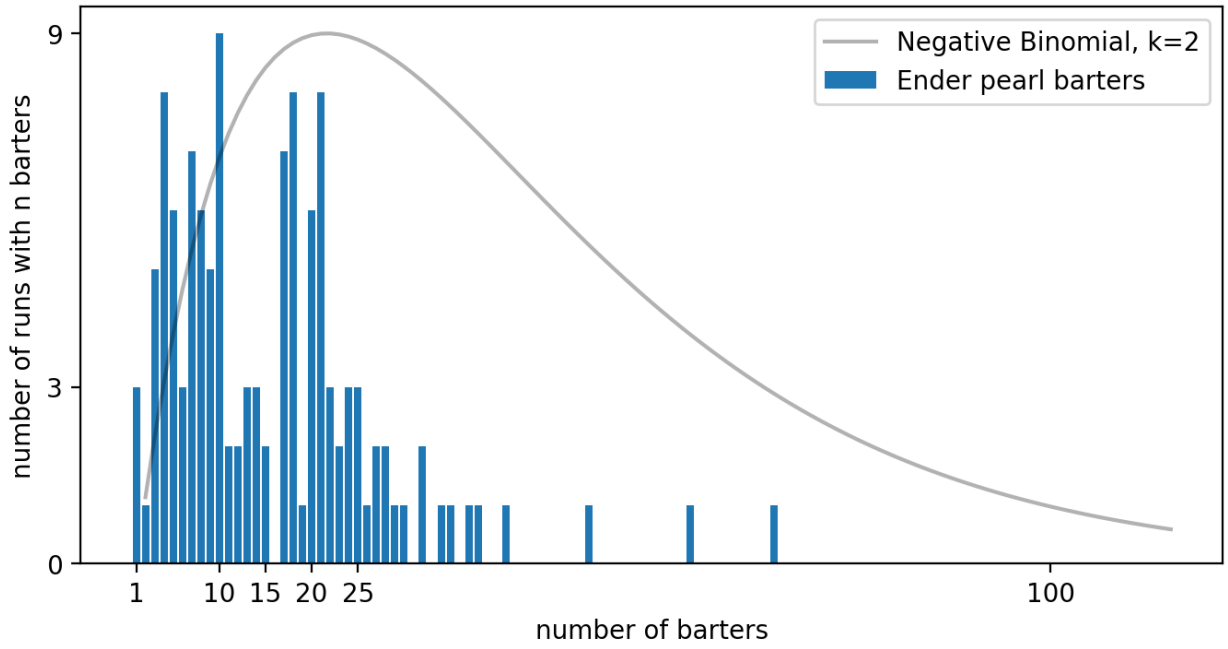

Fig. 4.4: The number of barters performed during a speedrun, for sessions that had zero to two successful barters, according to the Minecraft Speeedrunning Team.

(continued from previous page)

```
    pearl_y = [all_pearl_barters[ mask & (all_pearl_barters['n'] == n) ]['n'].count()␣
↪for n in pearl_x]

    # colour helps separate these
    colour    = [.3,.3,.3]
    colour[k] = .6
    axs[k].bar( pearl_x, pearl_y, label=f'k={k}', color=colour )

    # add the negative binomial
    nbinom_y = nbinom(2, r_pearl).pmf( x )
    axs[k].plot( x+2, nbinom_y/np.max(nbinom_y)*np.max(pearl_y), '-k', alpha=0.3 )

    axs[k].set_xticks([1,10,15,20,25,100])
    axs[k].set_yticks([0,3,6])
    axs[k].legend()

# we only need one label
axs[1].set_ylabel( 'frequency' )
axs[2].set_xlabel( 'number of barters in a run' )

fig_show( fig, 'fig:barters_by_k' )
```

The number of barters differs significantly for speedrunners that had zero Ender pearl barters. This makes sense if they are time or resource limited, as the odds of getting two successful barters before the limit is reached diminishes with each barter performed. Interestingly, the distribution bears a vague resemblance to the Exponential, implying that speedrunners have a fixed probability of quitting after each barter. The resemblance is not perfect, though, as the peak number of barters is $n = 4$ instead of $n = 1$, the latter being what we expect from the Exponential distribution. This suggests speedrunners wait a few barters to see if they'll earn any Ender pearls before considering dropping out. A better strategy would be to calculate when the probability of earning two pearl barters before running out of time or

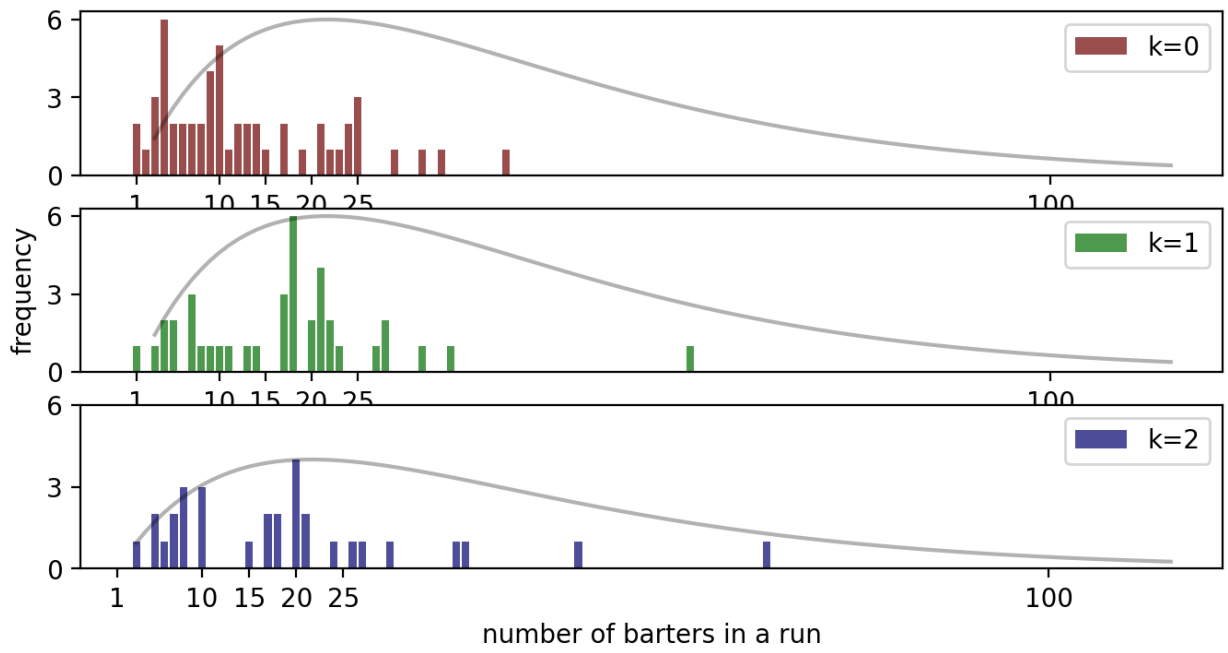

Fig. 4.5: The number of barters performed during a speedrun, for sessions that had zero to two successful barters, according to the Minecraft Speeedrunning Team. Each count of successful barters is plotted separately.

gold drops below a threshold and always bartering until that point, which gives a very different distribution.

The distributions for $k = 1$ and $k = 2$ are remarkably similar, though the former peaks earlier. This is more in line with the improved strategy outlined above, with further evidence coming from the lack of runs for $k = 1$ and $8 \leq n \leq 16$. There are not a lot of datapoints to work from, however, so these could be false patterns.

Our simulation for Ender pearl barters must take as much of the above into account as possible. I propose the following:

1. Draw a maximum number of barters to perform, n_{max}, from the observed distribution of n for $k < 3$.
2. Draw a number of successful barters, k_{max}, from $\text{Binom}(n_{\text{max}}, r)$, where $r \in \{r_{\text{fair}}, r_{\text{cheat}}\}$.
3. Draw a number of barters to perform, n_{success}, from $\text{NegBinom}(2, r)$.
4. If $n_{\text{success}} < n_{\text{max}}$, use $(n_{\text{success}}, 2)$ as the results of this run. Otherwise, use $(n_{\text{max}}, k_{\text{max}})$.

Notice that k is never checked, so $k_{\text{max}} > 2$ is possible. This is a better match for the observed data, as sometimes speedrunners do have more than two Ender pearl barters per run. For the cheating player, we will use $r_{\text{cheat}} \approx 6.383\%$. All values besides r_{fair} are identical with the previous simulation.

```
random  = np.random.default_rng(64)

r_fair    = 20/423 ; glue( 'sim_pearl_r_fair', r_fair*100, display=False )
r_cheat_h = 27/423 ; glue( 'sim_pearl_r_cheat_h', r_cheat_h*100, display=False )
count     = 200    ; glue( 'sim_pearl_count', count, display=False )
clones    = 16     ; glue( 'sim_pearl_clones', clones, display=False )
a_prior, b_prior   = prior(r_fair, 4)

# set up n_max
choices_n = np.array( all_pearl_barters[all_pearl_barters['k'] < 3]['n'].unique(),␣
↪dtype='int64' )
weights_n = [all_pearl_barters[(all_pearl_barters['k'] < 3) & (all_pearl_barters['n']␣
↪== n)]['n'].count() \
```

(continues on next page)

(continued from previous page)

```
            for n in choices_n]
weights_n = weights_n / np.sum(weights_n)

# draw n_max, k_max, and n_success for the cheater
vec_cheat_n_binom = random.choice( choices_n, p=weights_n, size=(clones,count) )
vec_cheat_k_binom = random.binomial( vec_cheat_n_binom, r_cheat_h )
vec_cheat_n_nbinom = random.negative_binomial( 2, r_cheat_h, size=(clones,count) ) + 2

# merge the two via vectorization
mask = vec_cheat_n_nbinom < vec_cheat_n_binom
vec_cheat_k = 2*mask + vec_cheat_k_binom*(1-mask)
vec_cheat_n = np.minimum( vec_cheat_n_nbinom, vec_cheat_n_binom )

# wash-rinse-repeat for the fair player
vec_fair_n_binom = random.choice( choices_n, p=weights_n, size=(clones,count) )
vec_fair_k_binom = random.binomial( vec_fair_n_binom, r_fair )
vec_fair_n_nbinom = random.negative_binomial( 2, r_fair, size=(clones,count) ) + 2

mask = vec_fair_n_nbinom < vec_fair_n_binom
vec_fair_k = 2*mask + vec_fair_k_binom*(1-mask)
vec_fair_n = np.minimum( vec_fair_n_nbinom, vec_fair_n_binom )

# calculate cumulative sums
sum_fair_k = np.cumsum( vec_fair_k, axis=1 )
sum_fair_n = np.cumsum( vec_fair_n, axis=1 )
sum_cheat_k = np.cumsum( vec_cheat_k, axis=1 )
sum_cheat_n = np.cumsum( vec_cheat_n, axis=1 )

fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

x = np.arange(1,count+1)
for idx in range(clones):
    y = [BF_H_fair_H_cheat(sum_cheat_k[idx][i], sum_cheat_n[idx][i], r_fair, a_prior,␣
↪b_prior) for i in range(count)]
    if idx == 0:
        plt.plot( x, y, '-r', label=f'cheater (r={r_cheat_h:.3f})' )
    else:
        plt.plot( x, y, '-r', alpha=0.1 )

    y = [BF_H_fair_H_cheat(sum_fair_k[idx][i], sum_fair_n[idx][i], r_fair, a_prior, b_
↪prior) for i in range(count)]
    if idx == 0:
        plt.plot( x, y, '-k', label=f'fair (r={r_fair:.3f})' )
    else:
        plt.plot( x, y, '-k', alpha=0.1 )

plt.plot( x, [1 for v in x], '--g', label='break even' )

plt.xlabel("rounds")
plt.xticks([1,count])
plt.ylabel('H_fair / H_cheat')
plt.yscale("log")

plt.legend()
fig_show( fig, 'fig:sim_pearl_duo' )
```

The *resulting simulation* has a number of interesting properties. The cheater shows a similar descent to *the previous*

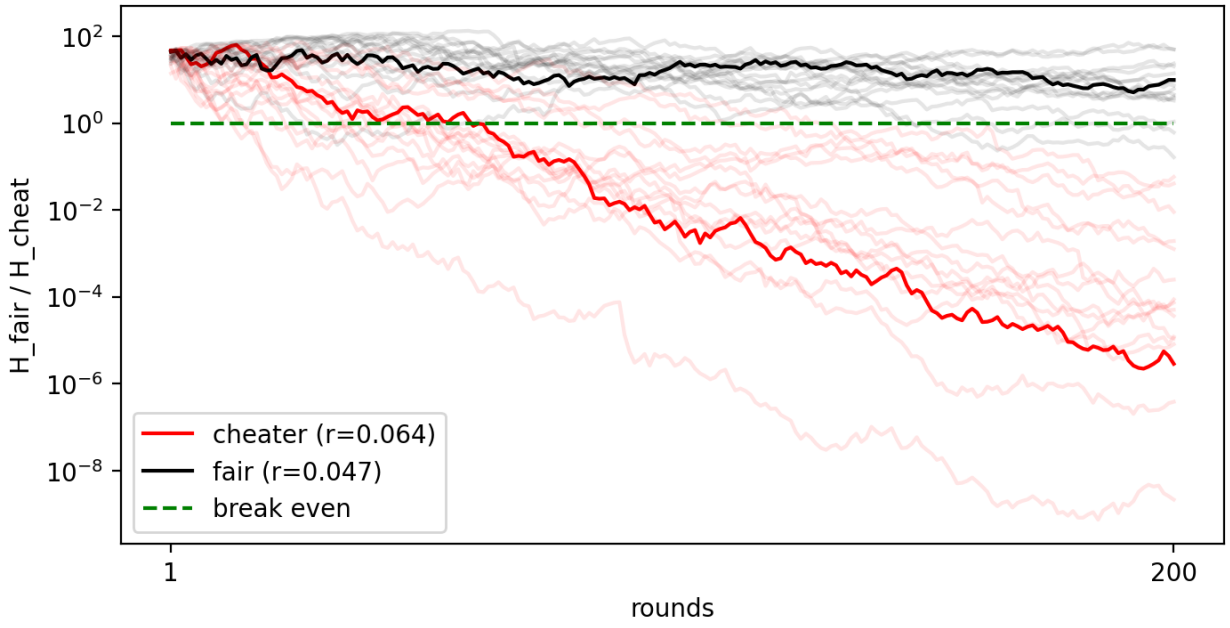

Fig. 4.6: The behaviour of this Bayes Factor for 16 clones of a player who modified their Ender pearl barter rate to be $r_{\text{cheat}} \approx 6.383\%$, versus clones of a player who has not ($r_{\text{fair}} \approx 4.728\%$).

figure, and yet as a proportion the level of cheating is more blatant. This is likely due to a combination of two things. Intuitively, it seems obvious that the rate of rare events would be more difficult to estimate than more common ones. As a consequence, though, it must also be difficult to estimate the frequency of *very common* events; if it was not, then we could switch from tracking when rare events do occur to when they do *not* and achieve a better estimate. This implies that the easiest event rate to track is neither common nor rare, when the rate of the event occurring matches the rate of it not occurring. That is $r = \frac{1}{2}$, the rate of Blaze rod drops, so by this argument the lower rate of Ender pearl barters must make them more difficult to estimate. This intuition can be placed on firmer ground via Information theory.

It also should be noted that my prior for Ender pearl barters is fairly strong. My scale 4 prior for Blaze rod drops is mathematically equivalent to a Bayes/Laplace prior with six "phantom" observations added that precisely match the expected drop rate. The scale 4 prior for Ender pearl drops is equivalent to Bayes/Laplace with three successful Ender pearl drops observed and slightly more than 81 failed attempts. With so many phantom observations, overcoming the latter prior's weight requires either many more observations or a greater level of bias.

Another interesting property is that the fair player no longer follows the $\sqrt{n}$ rule, and in fact their Bayes factors show a gradual decline over time. You might have assumed that since this simulation was the combination of Negative Binomial and Binomial distributions with the same rate, and the math showed that this Bayes factor worked equally well on both distributions, it should have worked for our simulation. In reality, the way we combined the two distributions together created a subtle bias in the underlying success rate, skewing it from $\frac{20}{423}$. The PE report was right to worry about this possibility, even if got the details wrong.

At the same time, the behaviour of the cheater and fair player are clearly different. A skew may be present, but it is much smaller than the difference between the cheating and fair rate. This shows that a flawed model can still be useful if it sufficiently close to reality, and if its flaws can be bounded. This is another reminder of why control groups and validation are so important.

CHAPTER

FIVE

REAL-WORLD RESULTS

With the above analysis in place, we're ready to look at real world data.

5.1 Blaze Rod Drops

We will begin with the Blaze rod data. The MST report gathered data for five distinct random-seed any-percent Minecraft 1.16 speedrunners: Benex, Dream, Illumina, Sizzler, and Vadikus007.

```
# Execute this cell to install all dependencies. Apologies for the spam.

# If you're running this on your own computer, I recommend altering "install mpmath"
#  to read "install --user mpmath", that way you don't need administrator access.
# You can get rid of the spam by changing "pip install" to "pip -q install", but on␣
↪Colab
#  you could miss a message that you need to restart your runtime. If you don't␣
↪restart,
#  the code won't work, hence why the default is to spam.
# A few error messages are fine, I usually get one about mismatched versions and the␣
↪code
#  still runs.

!pip install mpmath myst_nb numpy pandas matplotlib scipy
```

```
Requirement already satisfied: mpmath in /usr/lib/python3/dist-packages (1.1.0)
Requirement already satisfied: myst_nb in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (0.10.2)
```

```
Requirement already satisfied: numpy in /usr/lib/python3/dist-packages (1.18.4)
Requirement already satisfied: pandas in /usr/lib/python3/dist-packages (1.0.5)
Requirement already satisfied: matplotlib in /usr/lib/python3/dist-packages (3.3.0)
Requirement already satisfied: scipy in /usr/lib/python3/dist-packages (1.5.2)
Requirement already satisfied: sphinx-togglebutton~=0.2.2 in /home/hjhornbeck/.local/
↪lib/python3.8/site-packages (from myst_nb) (0.2.3)
Requirement already satisfied: pyyaml in /usr/lib/python3/dist-packages (from myst_
↪nb) (5.3.1)
```

```
Requirement already satisfied: docutils>=0.15 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from myst_nb) (0.16)
Requirement already satisfied: ipywidgets<8,>=7.0.0 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (7.6.3)
```

```
Requirement already satisfied: jupyter-sphinx==0.3.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (0.3.1)
Requirement already satisfied: nbformat~=5.0 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from myst_nb) (5.0.8)
```

```
Requirement already satisfied: importlib-metadata in /usr/lib/python3/dist-packages␣
↪(from myst_nb) (1.6.0)
Requirement already satisfied: ipython in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from myst_nb) (7.19.0)
```

```
Requirement already satisfied: sphinx<4,>=2 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from myst_nb) (3.4.3)
```

```
Requirement already satisfied: nbconvert~=5.6 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from myst_nb) (5.6.1)
```

```
Requirement already satisfied: jupyter-cache~=0.4.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (0.4.1)
```

```
Requirement already satisfied: myst-parser~=0.12.9 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (0.12.10)
```

```
Requirement already satisfied: setuptools in /usr/lib/python3/dist-packages (from␣
↪sphinx-togglebutton~=0.2.2->myst_nb) (49.3.1)
Requirement already satisfied: wheel in /usr/lib/python3/dist-packages (from sphinx-
↪togglebutton~=0.2.2->myst_nb) (0.34.2)
Requirement already satisfied: traitlets>=4.3.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from ipywidgets<8,>=7.0.0->myst_nb) (5.0.5)
Requirement already satisfied: widgetsnbextension~=3.5.0 in /home/hjhornbeck/.local/
↪lib/python3.8/site-packages (from ipywidgets<8,>=7.0.0->myst_nb) (3.5.1)
```

```
Requirement already satisfied: ipykernel>=4.5.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from ipywidgets<8,>=7.0.0->myst_nb) (5.4.2)
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >= "3.6" in /
↪home/hjhornbeck/.local/lib/python3.8/site-packages (from ipywidgets<8,>=7.0.0->myst_
↪nb) (1.0.0)
Requirement already satisfied: ipython-genutils in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from nbformat~=5.0->myst_nb) (0.2.0)
```

```
Requirement already satisfied: jupyter-core in /usr/lib/python3/dist-packages (from␣
↪nbformat~=5.0->myst_nb) (4.6.3)
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/lib/python3/dist-
↪packages (from nbformat~=5.0->myst_nb) (3.2.0)
Requirement already satisfied: backcall in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (0.2.0)
Requirement already satisfied: pygments in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from ipython->myst_nb) (2.7.3)
Requirement already satisfied: pexpect>4.3; sys_platform != "win32" in /usr/lib/
↪python3/dist-packages (from ipython->myst_nb) (4.6.0)
```

```
Requirement already satisfied: jedi>=0.10 in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (0.17.0)
Requirement already satisfied: pickleshare in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (0.7.5)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/
↪lib/python3/dist-packages (from ipython->myst_nb) (3.0.6)
Requirement already satisfied: decorator in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (4.4.2)
Requirement already satisfied: babel>=1.3 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from sphinx<4,>=2->myst_nb) (2.9.0)
Requirement already satisfied: sphinxcontrib-jsmath in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.1)
```

```
Requirement already satisfied: Jinja2>=2.3 in /usr/lib/python3/dist-packages (from␣
↪sphinx<4,>=2->myst_nb) (2.11.2)
Requirement already satisfied: sphinxcontrib-devhelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.2)
Requirement already satisfied: sphinxcontrib-qthelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.3)
```

```
Requirement already satisfied: sphinxcontrib-serializinghtml in /home/hjhornbeck/.
↪local/lib/python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.1.4)
Requirement already satisfied: requests>=2.5.0 in /usr/lib/python3/dist-packages␣
↪(from sphinx<4,>=2->myst_nb) (2.23.0)
Requirement already satisfied: alabaster<0.8,>=0.7 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (0.7.12)
Requirement already satisfied: packaging in /usr/lib/python3/dist-packages (from␣
↪sphinx<4,>=2->myst_nb) (20.4)
```

```
Requirement already satisfied: snowballstemmer>=1.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (2.0.0)
Requirement already satisfied: sphinxcontrib-htmlhelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.3)
Requirement already satisfied: sphinxcontrib-applehelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.2)
```

```
Requirement already satisfied: imagesize in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from sphinx<4,>=2->myst_nb) (1.2.0)
Requirement already satisfied: testpath in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (0.4.4)
Requirement already satisfied: defusedxml in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (0.6.0)
Requirement already satisfied: entrypoints>=0.2.2 in /usr/lib/python3/dist-packages␣
↪(from nbconvert~=5.6->myst_nb) (0.3)
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/lib/python3/dist-packages␣
↪(from nbconvert~=5.6->myst_nb) (0.8.4)
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/lib/python3/dist-packages␣
↪(from nbconvert~=5.6->myst_nb) (1.4.2)
Requirement already satisfied: bleach in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (3.2.1)
Requirement already satisfied: sqlalchemy~=1.3.12 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from jupyter-cache~=0.4.1->myst_nb) (1.3.22)
```

```
Requirement already satisfied: attrs in /usr/lib/python3/dist-packages (from jupyter-
↪cache~=0.4.1->myst_nb) (19.3.0)
```

```
Requirement already satisfied: nbdime in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from jupyter-cache~=0.4.1->myst_nb) (2.1.0)
```

```
Requirement already satisfied: nbclient<0.6,>=0.2 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from jupyter-cache~=0.4.1->myst_nb) (0.5.1)
```

```
Requirement already satisfied: markdown-it-py~=0.5.4 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst-parser~=0.12.9->myst_nb) (0.5.8)
```

```
Requirement already satisfied: notebook>=4.4.1 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from widgetsnbextension~=3.5.0->ipywidgets<8,>=7.0.0->myst_nb) (6.
↪1.6)
```

```
Requirement already satisfied: tornado>=4.2 in /usr/lib/python3/dist-packages (from␣
↪ipykernel>=4.5.1->ipywidgets<8,>=7.0.0->myst_nb) (6.0.4)
Requirement already satisfied: jupyter-client in /usr/lib/python3/dist-packages (from␣
↪ipykernel>=4.5.1->ipywidgets<8,>=7.0.0->myst_nb) (6.1.6)
Requirement already satisfied: pytz>=2015.7 in /usr/lib/python3/dist-packages (from␣
↪babel>=1.3->sphinx<4,>=2->myst_nb) (2020.1)
```

```
Requirement already satisfied: colorama in /usr/lib/python3/dist-packages (from␣
↪nbdime->jupyter-cache~=0.4.1->myst_nb) (0.4.3)
Requirement already satisfied: GitPython!=2.1.4,!=2.1.5,!=2.1.6 in /home/hjhornbeck/.
↪local/lib/python3.8/site-packages (from nbdime->jupyter-cache~=0.4.1->myst_nb) (3.1.
↪12)
```

```
Requirement already satisfied: six in /usr/lib/python3/dist-packages (from nbdime->
↪jupyter-cache~=0.4.1->myst_nb) (1.15.0)
Requirement already satisfied: nest-asyncio in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from nbclient<0.6,>=0.2->jupyter-cache~=0.4.1->myst_nb) (1.4.3)
Requirement already satisfied: async-generator in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from nbclient<0.6,>=0.2->jupyter-cache~=0.4.1->myst_nb) (1.10)
Requirement already satisfied: argon2-cffi in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets<8,>=7.0.
↪0->myst_nb) (20.1.0)
```

```
Requirement already satisfied: Send2Trash in /usr/lib/python3/dist-packages (from␣
↪notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets<8,>=7.0.0->myst_nb) (1.5.0)
Requirement already satisfied: terminado>=0.8.3 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets
↪<8,>=7.0.0->myst_nb) (0.9.2)
Requirement already satisfied: pyzmq>=17 in /usr/lib/python3/dist-packages (from␣
↪notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets<8,>=7.0.0->myst_nb) (19.0.2)
```

```
Requirement already satisfied: prometheus-client in /usr/lib/python3/dist-packages␣
↪(from notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets<8,>=7.0.0->myst_nb) (0.
↪7.1)
Requirement already satisfied: gitdb<5,>=4.0.1 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from GitPython!=2.1.4,!=2.1.5,!=2.1.6->nbdime->jupyter-cache~=0.4.
↪1->myst_nb) (4.0.5)
Requirement already satisfied: cffi>=1.0.0 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from argon2-cffi->notebook>=4.4.1->widgetsnbextension~=3.5.0->
↪ipywidgets<8,>=7.0.0->myst_nb) (1.14.4)
Requirement already satisfied: ptyprocess; os_name != "nt" in /home/hjhornbeck/.local/
↪lib/python3.8/site-packages (from terminado>=0.8.3->notebook>=4.4.1->
↪widgetsnbextension~=3.5.0->ipywidgets<8,>=7.0.0->myst_nb) (0.7.0)
Requirement already satisfied: smmap<4,>=3.0.1 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from gitdb<5,>=4.0.1->GitPython!=2.1.4,!=2.1.5,!=2.1.6->nbdime->
↪jupyter-cache~=0.4.1->myst_nb) (3.0.4)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: pycparser in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from cffi>=1.0.0->argon2-cffi->notebook>=4.4.1->widgetsnbextension~
↪=3.5.0->ipywidgets<8,>=7.0.0->myst_nb) (2.20)
```

```
from fractions import Fraction

from math import log,factorial
import matplotlib.pyplot as plt
from mpmath import mp
from myst_nb import glue

import numpy as np

import pandas as pd
from scipy.optimize import differential_evolution
from scipy.stats import beta,binom,nbinom


dpi         = 200  # change this to increase/decrease the resolution charts are made␣
↪at
book_output = True ### changing this to False will allow you to view the plots in a␣
↪Jupyter notebook

def fig_show( fig, name ):
    """A helper to control how we're displaying figures."""
    global book_output

    if book_output:
        glue( name, fig, display=False )
        plt.close();
    else:
        fig.show()
```

```
# missing "simple_bernoulli_bf.py"? Uncomment this line to grab it from the repository
# !wget -c "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_cheating/main/
↪simple_bernoulli_bf.py"

from simple_bernoulli_bf import prior, posterior_H_fair, BF_H_fair_H_cheat
```

```
# missing the data files? Uncommenting and running this cell might retrieve them

# !mkdir data
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.benex.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.dream.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.illumina.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.sizzler.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.vadikus007.tsv"
```

```
colours = ['b', 'g', 'r', 'c', 'm', 'y', 'k']  # makes colouring some of the␣
↪following easier
```

(continues on next page)

(continued from previous page)

```
colours = colours + colours                    # double-up to allow for extra data

blaze_players = ['benex','dream','illumina','sizzler','vadikus007']

blaze_rods = {p:pd.read_csv(f'data/blaze.{p}.tsv',sep="\t") for p in blaze_players}


# Want to add your own data? The easiest way is to gather your data, then edit and␣
↪uncomment this:

# vec_n = [12, 11, 16, 10, 13, 11, 5, 4, 1, 14]
# vec_k = [ 6,  7,  8,  7,  8,  8, 5, 3, 1,  8]
# blaze_rods['NAME'] = pd.DataFrame( {'n':vec_n, 'k':vec_k} )

blaze_rods['dream'].set_index('n').transpose()
```

```
n  12  11  16  10  13  11  5   4   1   14  ...  9   10  15  10  15  11  9   \
k   6   7   8   7   8   8   5   3   1   8  ...   8   6   8   7   7   7   7

n  10  9   9
k   7   7   8

[1 rows x 33 columns]
```

We'll use the same charts as with the simulation, where the Bayes factor is updated as new data is added. This responds to a critique of the PE report: if the same trend is observed regardless of whether or not the last datapoint is dropped, we can be reasonably confident it was not an artifact of our stopping point.

Against Weak Priors

Why don't we automatically choose the weakest prior? One reason is that the data usually exhibits some random variance, and it's possible that the small bit of data we're looking at happens to be an outlier. A strong prior will have more "inertia" than a weaker one, and therefore be more resilient to outliers. If the data is genuinely outside of where the prior expects, it should tug the posterior into place if we keep piling on the data.

As promised, we will determine the influence of the prior by varying the scale parameter. My preferred value of 4 will be scaled by values ranging between $\frac{1}{3}$ and 3. The higher multiplier will "tug" the posterior towards the default hypothesis, while the lower gives the data more power to shape the posterior.

```
r_fair = Fraction(1,2)

# only the two extremes need to be explicitly calculated
priors = [prior( r_fair, scale ) for scale in [Fraction(4,3), 4, 12]]


fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

max_count = 0  # note the largest number of runs by a player
xtics = set()  # we'd like some fancy X-axis ticks

for i,p in enumerate(blaze_rods.keys()):

    # update the above variables
    count = len(blaze_rods[p])
```

(continues on next page)

(continued from previous page)

```
    if count > max_count:
        max_count = count
    xtics.add( count )

    x     = np.arange(1, count+1)
    sum_n = np.cumsum( blaze_rods[p]['n'] )
    sum_k = np.cumsum( blaze_rods[p]['k'] )

    if p == 'dream':
        alpha = 1
    else:
        alpha = 0.3

    y = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[1]) for i in␣
↪range(count)]
    plt.plot( x, y, '-', label=p, c=colours[i], alpha=alpha )

    y_low = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[0]) for i in␣
↪range(count)]
    y_hig = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[2]) for i in␣
↪range(count)]
    plt.fill_between( x, [float(v) for v in y_low], [float(v) for v in y_hig], \
                      alpha=0.2*alpha, color=colours[i] )

# build the fancy ticks
xtics.discard( max_count )
xtics = [ (x >> 1) << 1 for x in xtics]
xtics.extend( [ ((x >> 1) + 1) << 1 for x in xtics if x + 1 < max_count] )
xtics.extend( [1,max_count] )


x = range(1,max_count+1)
plt.plot( x, [1 for v in x], '--k', label='break even' )

plt.xlabel("run")
plt.xticks( list(xtics) )
plt.ylabel('H_fair / H_cheat')
plt.yscale("log")

plt.legend()
fig_show( fig, 'fig:data_blaze' )
```

The Bayes factor for Dream's Blaze rod drop rate *behaves differently* from other Minecraft speedrunners. It drops like the cheating players from *the Blaze rod simulation*, but at a much sharper rate. Removing the last datapoint does not impact that trend. There is evidence the Bayes factor of two other runners are curving back towards the 1:1 odds line, but that's still consistent with random fluctuations due to normal gameplay. It might also indicate Blaze rods do not follow the Binomial or Negative Binomial distributions, but we'd need quite a bit more data to be sure.

The choice of prior doesn't have a significant impact, but it should be noted the log scaling downplays the difference between the priors. The Bayes factor for Dream's round thirty-three using the scale 4 prior is half the size of the BF for the scale 12 prior, for instance.

```
r_fair         = Fraction(1,2)
sum_n          = blaze_rods['dream']['n'].sum()
sum_k          = blaze_rods['dream']['k'].sum()
dream_blaze_bf = BF_H_fair_H_cheat( sum_k, sum_n, r_fair, *prior(r_fair,4) )
```

(continues on next page)

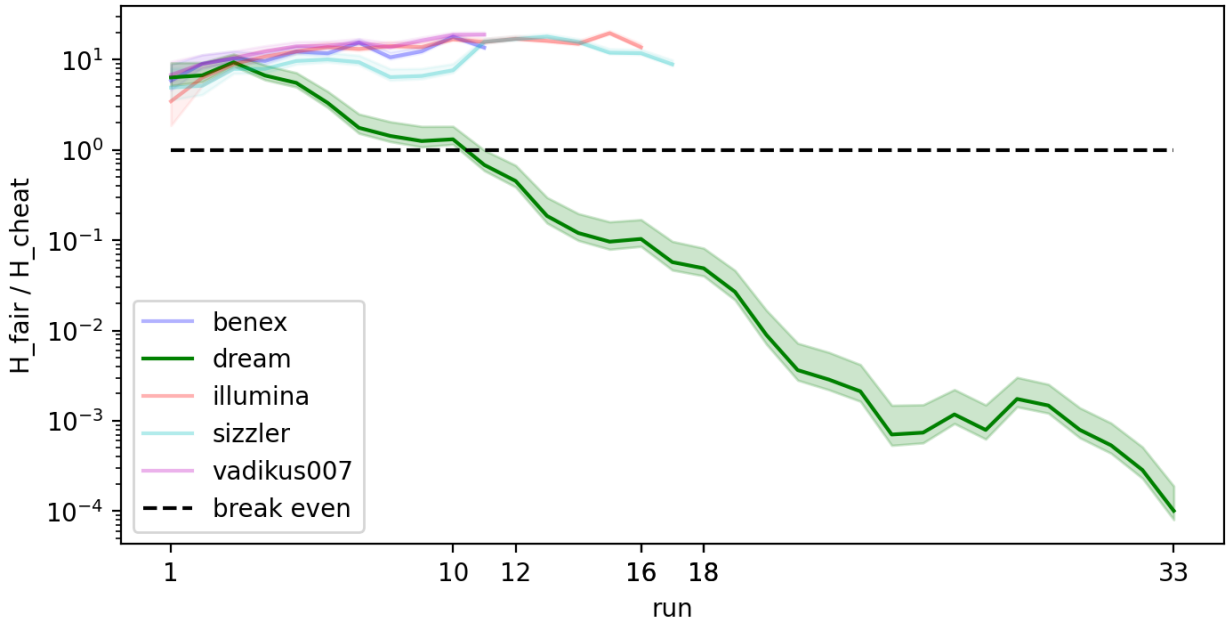

Fig. 5.1: The behaviour of this Bayes Factor for the Blaze rod drops of serveral Minecraft 1.16 speedrunners, using data from the MST report.

(continued from previous page)

```
print( f"The Bayes factor for Dream's Blaze rod rate, over all his rounds and with␣
↪the scale 4 prior," + \
       f" is about {mp.nstr(dream_blaze_bf,3)}", end='' )

if dream_blaze_bf < 1:
    print( f", or 1:{int(mp.fdiv(1,dream_blaze_bf)):,}." )
else:
    print( "." )
```

```
The Bayes factor for Dream's Blaze rod rate, over all his rounds and with the scale 4␣
↪prior, is about 0.000101, or 1:9,930.
```

The resulting Bayes factor is roughly equivalent to the average golfer landing a hole in one on their next shot. In comparison, the equivalent p-value in the MST report is 4.72×10^{-11}. Setting aside the different interpretations of both numbers, that's a difference of seven orders of magnitude. This seems to bear out *the assertion* that Bayesian statistics is inherently more sluggish to respond to the evidence.

The PE report puts the probability of fair play at 3×10^{-8}, however. Why do two Bayesian analyses come to two very different results? The PE report analyzed Dream's drop rate by calculating a grid of probabilities ranging from $\frac{1}{2}$ to $\frac{9}{10}$, then taking the result for $r_{\text{blaze}} = \frac{1}{2}$ as the probability of fair play (pg. 10). This is analogous to *the idealized dart board*, which by necessity will lead to more extreme values. By integrating over an area, my definition of both H_{fair} and H_{cheat} account for the random fluctuations we would expect from random data, which necessarily waters down the probabilities of each. The PE report also does not factor in the probability of H_{cheat}. If that hypothesis is unlikely, then the relative likelihood of H_{fair} is boosted proportionately.

As nice as *the Bayes factor chart* is, it only tells us the relative probabilities of H_{fair} to H_{cheat}. It gives no indication of what the most credible drop rate was. We can derive that by charting the posteriors for each player and noting the mean values. I'll again vary the prior to observe how the posterior changes.

```
r_fair = Fraction(1,2)

fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

dream_mean = 0

# we can't get away with just doing the extremes. Instead, spam priors!
priors = [prior( r_fair, scale ) for scale in np.exp(np.linspace( np.log(4/3), np.
↪log(12), 32 ))]
a_prior, b_prior = prior( r_fair, 4 )

x_limits = [.4, .8]
x = np.linspace( *x_limits, 512 )
for i,p in enumerate(blaze_rods.keys()):

    sum_n = np.sum( blaze_rods[p]['n'] )
    sum_k = np.sum( blaze_rods[p]['k'] )

    if p == 'dream':
        alpha = 1
    else:
        alpha = 0.3

    plt.plot( x, beta.pdf( x, float(a_prior + sum_k), float(b_prior + sum_n - sum_k)␣
↪), \
              '-', label=p, c=colours[i], alpha=alpha )
    for pr in priors:
        plt.plot( x, beta.pdf( x, float(pr[0] + sum_k), float(pr[1] + sum_n - sum_k)␣
↪), \
              '-', alpha=0.1*alpha, c=colours[i] )

    # plot the mean
    plt.axvline( (a_prior + sum_k)/(a_prior + b_prior + sum_n), c=colours[i], alpha=0.
↪3*alpha )

    if p == 'dream':
        dream_mean = (a_prior + sum_k)/(a_prior + b_prior + sum_n)

plt.xlabel("rate")
plt.xlim( x_limits )
plt.xticks( [.4, .5, .55, float(dream_mean), .8] )
plt.ylabel('likelihood')
plt.yticks([])

plt.legend()
fig_show( fig, 'fig:data_blaze_pos' )
```

The posterior distributions of other Minecraft players cluster on the generous side of 50:50 odds. Sizzler is the largest outlier, with the mean of his posterior approaching 55%, but even in their case a substantial amount of credence is present for $r_{\text{blaze}} \leq \frac{1}{2}$. Dream again stands alone, with the posterior's mean drop rate above $\frac{2}{3}$ and almost no credence around $\frac{1}{2}$. This is similar to the PE report's estimate of "around 0.7" (pg. 10). Strengthening the prior clusters the credence more tightly around $r_{\text{blaze}} = \frac{1}{2}$, but there is sufficient data to make the effect subtle at best.

```
r_fair           = Fraction(1,2)
sum_n            = blaze_rods['dream']['n'].sum()
sum_k            = blaze_rods['dream']['k'].sum()
a_prior, b_prior = prior(r_fair,4)
```

(continues on next page)

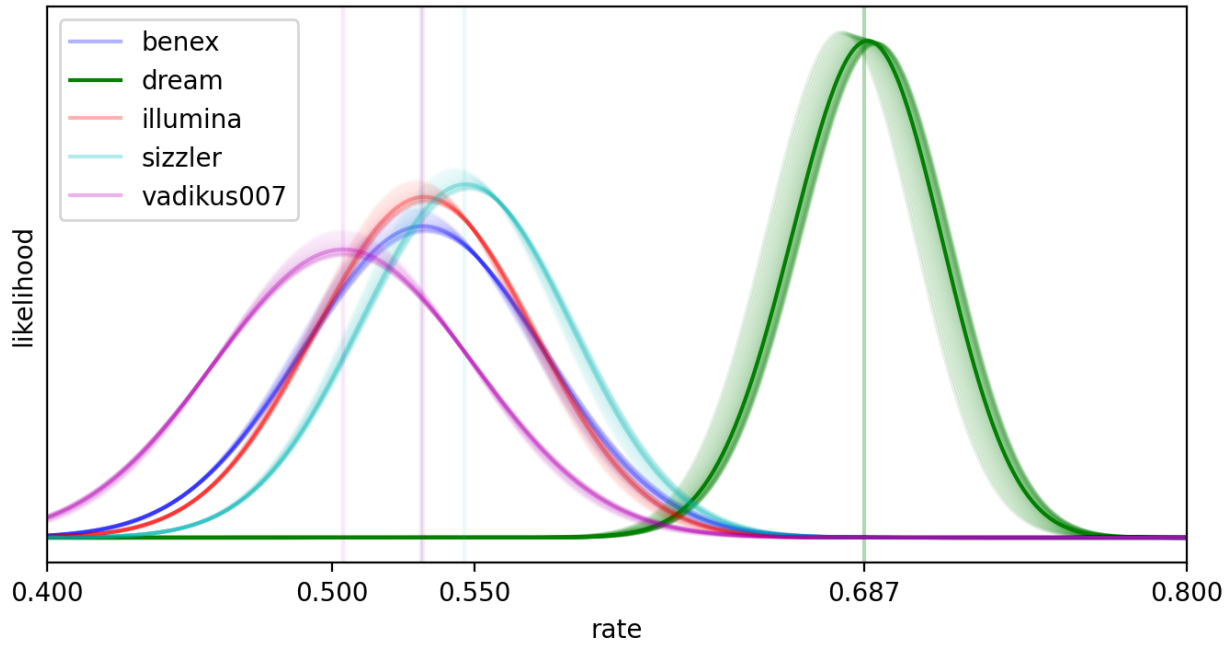

Fig. 5.2: The posterior distribution for the Blaze rod drops of several Minecraft 1.16 speedrunners, using data from the MST report. Vertical lines represent the mean of the posterior with the scale 4 prior.",

(continued from previous page)

```
bounds = [.025, .16, .5, .84, .975]
intervals = beta.ppf( bounds, float(a_prior + sum_k), float(b_prior + sum_n - sum_k) )
glue( 'data_blaze_intervals_low', intervals[0], display=False )
glue( 'data_blaze_intervals_high', intervals[4], display=False )
glue( 'interval_width', (bounds[4]-bounds[0])*100, display=False )

print( "Given the above data and prior, about two-thirds of our credence for Dream's␣
↪Blaze rod rate\n" + \
      f"  is in the interval [{intervals[1]:.3f}, {intervals[3]:.3f}], while about 95
↪% is between " + \
      f"[{intervals[0]:.3f}, {intervals[4]:.3f}]. The median is roughly
↪{intervals[2]:.3f}." )
```

```
Given the above data and prior, about two-thirds of our credence for Dream's Blaze␣
↪rod rate
  is in the interval [0.661, 0.713], while about 95% is between [0.635, 0.737]. The␣
↪median is roughly 0.687.
```

When estimating a parameter, it's quite common to generate confidence or credible intervals for it. Those can reiterate some of the problems of p-values, such as what the interval means or what counts as extreme,[MHR+16] plus there is no hard and fast rule on how much credence should be contained within the interval. The traditional value of 95% can be sensitive to extreme data, so I prefer $\frac{2}{3}$. If you insist on some sort of credible interval calculation, then the 95.0% credible interval for Dream's Blaze rod drop rate is [0.635, 0.737].

5.2 Ender Pearl Barters

We have two sources of data for Ender pearls. The MST report collected bartering data for five runners including Dream, and Dream himself provided a spreadsheet with more data. Accessing that spreadsheet requires giving Dream or a third party my Google account information, so I instead opted to use the data listed in Appendix A of the PE report. I've labelled this second source "dream_before", as it covers a period before Dream took time off of Minecraft speed runs. The MST's data on Dream is labelled "dream_after", for similar reasons.

This provides us with an easy way to consider which, if any, of *the above cheating scenarios* apply.

1. If Dream cheated, and if he altered his barter rate from the start, we would expect the Bayes factor of both datasets to act like cheating players from the simulation and both posteriors to have little credence around $r_{\text{blaze}} = \frac{20}{423}$.
2. If Dream cheated, but only after he took some time off, we would expect the BF of the "before" set to behave like any other Minecraft speedrunner while the BF of the "after" acts like a cheating player. The posteriors would, respectively, cluster around $\frac{20}{423}$ and higher than it.
3. If Dream cheated, but craftily reduced his barter rate to hide it, we would expect the BF of both datasets to act like cheating players. The posteriors would, respectively, cluster below $\frac{20}{423}$ and above it.

Scenario 1 has already been ruled out by the PE report, as it notes the additional data reduces the probability of Dream cheating (pg. 13).

```
# missing the data files? Uncommenting and running this cell might retrieve them

# !mkdir data
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.benex.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.dream_before.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.dream_after.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.illumina.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.sizzler.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.vadikus007.tsv"
```

```
colours = ['b', 'g', 'r', 'c', 'm', 'y', 'k']  # makes colouring some of the␣
↪following easier
colours = colours + colours                    # double-up to allow for extra data

pearl_players = ['benex','dream_before','dream_after','illumina','sizzler','vadikus007
↪']

bartering  = {p:pd.read_csv(f'data/bartering.{p}.tsv',sep="\t") for p in pearl_
↪players}


# Want to add your own data? The easiest way is to gather your data, then edit and␣
↪uncomment this:

# vec_n = [12, 11, 16, 10, 13, 11, 5, 4, 1, 14]
# vec_k = [ 6,  7,  8,  7,  8,  8, 5, 3, 1,  8]
# bartering['NAME'] = pd.DataFrame( {'n':vec_n, 'k':vec_k} )
```

(continues on next page)

(continued from previous page)

```
bartering['dream_after'].set_index('n').transpose()
```

```
n  22  5   24  18  4   1   7   12  26  8   ...  2   13  10  10  21  20  10  \
k   3   2   2   2   0   1   2   5   3   2  ...   0   1   2   2   2   2   2

n  3   18  3
k   1   2   2

[1 rows x 22 columns]
```

```
r_fair = Fraction(20,423)
priors = [prior( r_fair, scale ) for scale in [Fraction(4,3), 4, 12]]

# same old, same old
fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

max_count = 0
xtics = set()

for i,p in enumerate(bartering.keys()):

    count = len(bartering[p])
    if count > max_count:
        max_count = count
    xtics.add( count )

    x     = np.arange(1, count+1)
    sum_n = np.cumsum( bartering[p]['n'] )
    sum_k = np.cumsum( bartering[p]['k'] )

    if p == 'dream_after':
        alpha = 1
    else:
        alpha = 0.3

    y = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[1]) for i in␣
↪range(count)]
    plt.plot( x, y, '-', label=p, c=colours[i], alpha=alpha )

    y_low = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[0]) for i in␣
↪range(count)]
    y_hig = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[2]) for i in␣
↪range(count)]
    plt.fill_between( x, [float(v) for v in y_low], [float(v) for v in y_hig], \
                      alpha=0.1*alpha, color=colours[i] )

# build the fancy ticks
xtics.discard( max_count )
xtics = [ (x >> 1) << 1 for x in xtics]
xtics.extend( [ ((x >> 1) + 1) << 1 for x in xtics if x + 1 < max_count] )
xtics.extend( [1,max_count] )


x = range(1,max_count+1)
plt.plot( x, [1 for v in x], '--k', label='break even' )
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("run")
plt.xticks( xtics )

plt.ylabel('H_fair / H_cheat')
plt.yscale("log")

plt.legend()
fig_show( fig, 'fig:data_pearl' )
```

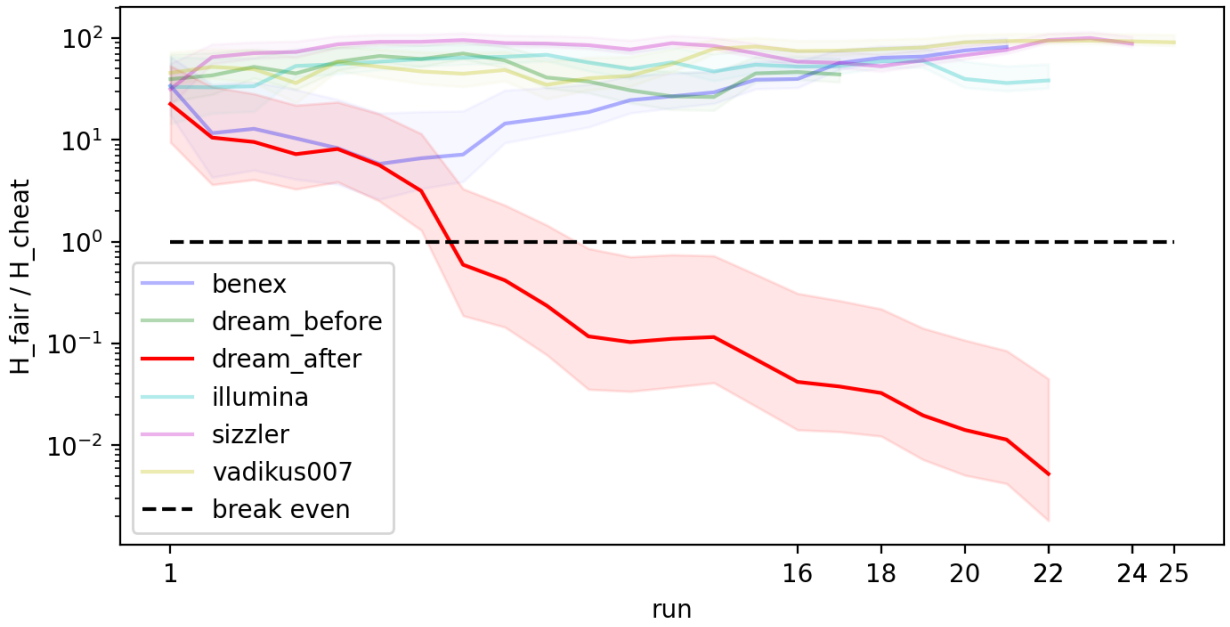

Fig. 5.3: The behaviour of this Bayes Factor for Ender pearl barters, from both reports.

This time there's less certainty. The same pattern is present, but only for Dream after he returned to random-seed any-percent Minecraft 1.16. The other speedrunners and Dream's earlier runs behave more like fair players, and generally follow the $\sqrt{n}$ curve. Benex's first six rounds do show a downward trajectory, but their later data restores the behaviour of a fair player. This provides mild evidence for Scenario 3 cheating, but the magnitudes involved are small enough that it can also be explained by unusually good luck. On the whole, for Dream, Scenario 2 is the most likely case.

The choice of prior has a much stronger on the final outcome here than it did *before*, and the probabilities are about two orders of magnitude higher. *As mentioned*, the rarity of Ender pearl barters relative to Blaze rod drops makes their rate harder to estimate, and the stronger prior for barters gives it a greater influence over the posterior.

```
sum_n          = np.sum( bartering['dream_after']['n'] )
sum_k          = np.sum( bartering['dream_after']['k'] )
dream_pearl_bf = BF_H_fair_H_cheat( sum_k, sum_n, r_fair, *prior(r_fair,4) )

print( f"The Bayes factor for Dream's post-break Ender pearl barters, over all his␣
↪rounds and with the scale 4 prior," + \
       f" is {mp.nstr(dream_pearl_bf,3)}", end='' )
if dream_pearl_bf < 1:
    print( f", or about 1:{int(mp.fdiv(1,dream_pearl_bf)):,}." )
else:
    print( "." )
```

```
The Bayes factor for Dream's post-break Ender pearl barters, over all his rounds and␣
↪with the scale 4 prior, is 0.00523, or about 1:191.
```

With my scale 4 prior, the relative odds between H_{fair} and H_{cheat} are about the odds of a US citizen dying of an accidental poisoning. Alternatively, it's more or less the odds of being born sometime during the first two days of June. A rare occurrence, certainly, but not that rare.

For comparison, the MST report put the p-value of this at 8.04×10^{-7}, which differs by five orders of magnitude. The PE report gives the probability of fair play at 3×10^{-10} (pg. 11), but that number uses the same methodology as was used for Blaze rods and so the same criticisms apply.

```
fig = plt.figure(figsize=(8, 4), dpi=dpi, facecolor='w', edgecolor='k')

dream_mean = 0  # store Dream's mean value here
dream_means = list()

# fill_between doesn't give good output here. Instead, spam priors!
priors = [prior( r_fair, scale ) for scale in np.exp(np.linspace( np.log(4/3), np.
↪log(12), 128 ))]
a_prior, b_prior = prior( r_fair, 4 )

x = np.linspace( 0, .2, 512 )
for i,p in enumerate(bartering.keys()):

    sum_n = np.sum( bartering[p]['n'] )
    sum_k = np.sum( bartering[p]['k'] )

    if p == 'dream_after':
        alpha = 1
    else:
        alpha = 0.3

    plt.plot( x, beta.pdf( x, float(a_prior + sum_k), float(b_prior + sum_n - sum_k)␣
↪), \
              '-', label=p, c=colours[i], alpha=alpha )
    for pr in priors:
        plt.plot( x, beta.pdf( x, float(pr[0] + sum_k), float(pr[1] + sum_n - sum_k)␣
↪), \
              '-', alpha=0.03*alpha, c=colours[i] )
        if p == 'dream_after':
            dream_means.append( (pr[0] + sum_k) / (pr[0] + pr[1] + sum_n) )

    plt.axvline( (a_prior + sum_k)/(a_prior + b_prior + sum_n), c=colours[i], alpha=0.
↪3 )

    if p == 'dream_after':
        dream_mean = (a_prior + sum_k)/(a_prior + b_prior + sum_n)

plt.xlabel("rate")
plt.xlim( [0,.2] )
plt.xticks( [0, float(r_fair), float(dream_mean), .2] )
plt.ylabel('likelihood')
plt.yticks([])

plt.legend()
fig_show( fig, 'fig:data_pearl_pos' )
```

All but one sequence hovers around the expected rate of Ender pearl barters. The posterior for Dream's performance

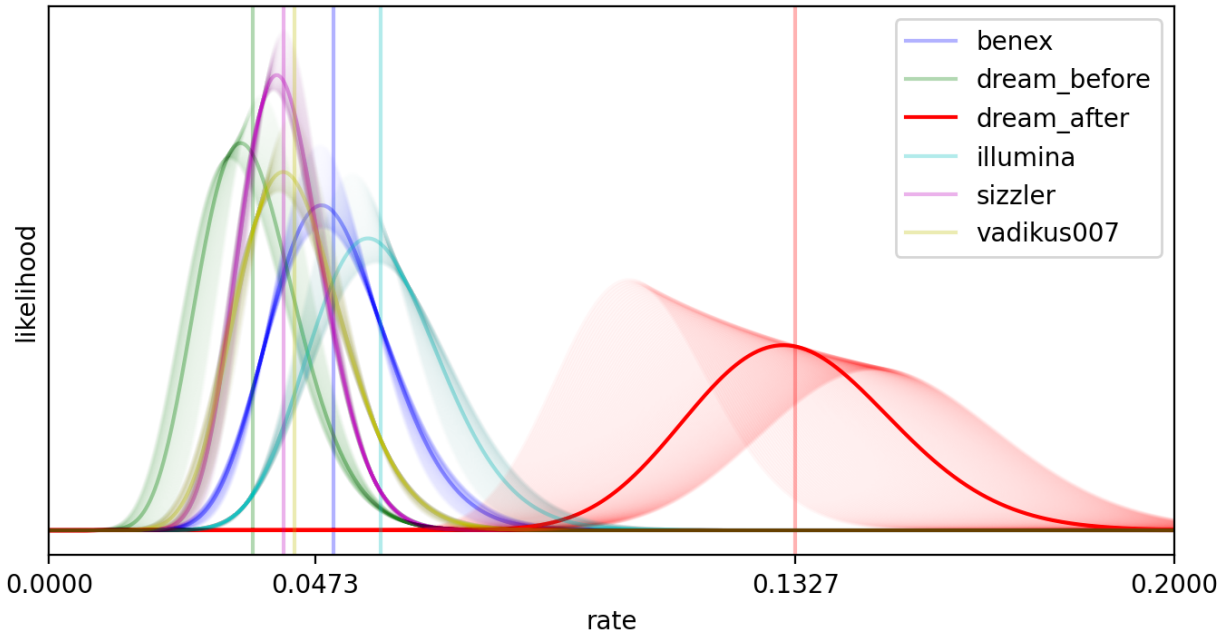

Fig. 5.4: The posterior distribution for the Ender pearl barters of several Minecraft 1.16 speedrunners, using data from both reports. Vertical lines represent the mean of the posterior for the scale 4 prior, and the effect of different priors is shown.

before his break favours lower barter rates than all other runners, providing a little support for Scenario 3, but a non-trivial amount of credence is greater than the expected barter rate. Overall, Scenario 2 remains more likely. Dream's post-break performance again is quite different from other speedrunners and even his own pre-break performance, but this time prior strength is a major factor. Even with the strongest prior, though, very little credence for Dream's post-break barters is around the unaltered rate.

The mean value of Dream's post-break performance is approximately three times larger than the expected rate, which again matches what the PE report observed (pg. 11).

```
r_fair           = Fraction(20,423)
sum_n            = bartering['dream_after']['n'].sum()
sum_k            = bartering['dream_after']['k'].sum()
a_prior, b_prior = prior(r_fair,4)

bounds = [.025, .16, .5, .84, .975]
intervals = beta.ppf( bounds, float(a_prior + sum_k), float(b_prior + sum_n - sum_k) )
glue( 'data_pearl_intervals_low', intervals[0], display=False )
glue( 'data_pearl_intervals_high', intervals[4], display=False )
glue( 'r_pearl', float(r_fair), display=False )

print( "Given the above data and prior, about two-thirds of our credence for Dream's␣
↪Ender pearl barter rate\n" + \
      f"  is in the interval [{intervals[1]:.3f}, {intervals[3]:.3f}], while about 95
↪% is between " + \
      f"[{intervals[0]:.3f}, {intervals[4]:.3f}]. The median is roughly
↪{intervals[2]:.3f}." )
```

```
Given the above data and prior, about two-thirds of our credence for Dream's Ender␣
↪pearl barter rate
```

(continues on next page)

(continued from previous page)

```
  is in the interval [0.115, 0.151], while about 95% is between [0.099, 0.170]. The␣
↪median is roughly 0.132.
```

A common heuristic is to see if the value we expect to see is contained within the interval, and invoke modus tollens if it is not. I do not endorse that, but if you disagree then I'll simply note that the 95.0% credible interval is [0.0991, 0.1703], which does not include the unaltered Ender pearl barter rate of 0.0473%.

5.3 Combining Both Datasets

As mentioned earlier, Bayes factors are likelihoods and can thus be multiplied together. This allows us to generate a combined odds ratio for both datasets.

```
dream_combo_bf = mp.fmul( dream_blaze_bf, dream_pearl_bf )

print( f"The Bayes factor for both rates, over all of Dream's post-break rounds and␣
↪with the scale 4 prior," + \
       f" is {mp.nstr(dream_combo_bf,3)}, or about 1:{int(mp.fdiv(1,dream_combo_bf)):,
↪}." )
```

```
The Bayes factor for both rates, over all of Dream's post-break rounds and with the␣
↪scale 4 prior, is 5.27e-7, or about 1:1,899,071.
```

The combined Bayes factor is quite low. To take an example, the median age of a Canadian in 2019 was about 41 years old. If we examine the relevant mortuary tables and extrapolate the odds of dying per year to the odds of dying per day, then we'd place four times more credence on a specific 41-year-old Canadian woman dying within 24 hours (about 1:472,632) than we would for H_{cheat} over H_{fair}.

As unlikely as that is, it's still seven orders of magnitude higher than the MST report's "loose upper bound" on Dream's success, which they declare is "almost certainly an overestimate" (pg. 22). After applying multiple corrections, the PE report concludes "there is a 1 in 100 million chance that a livestream in the Minecraft speedrunning community got as lucky this year on two separate random modes as Dream did in these six streams" (pg. 16), a number which is just about two order of magnitudes lower than what I calculate. The differences are entirely due to statistical methodology.

CHAPTER

SIX

CONCLUSION

I will refrain from stating conclusively that Dream boosted his Blaze rod drop and Ender pearl barter rate. To explain why, it is worth discussing stopping rules.

6.1 Stopping Rules

Both the MST and PE reports invoke stopping rules, though in a way that is not congruent with how they are used within the scientific community.

P-values

As I mentioned in *P-values are Not What You Think*, even professionals misunderstand p-values on a regular basis.

> Suppose that a researcher is testing whether one variable influences another or is comparing two treatments or effect sizes. Suppose also that the researcher is primarily interested in whether there is an effect (or difference) and, if so, the direction of the effect. Finally, suppose that the researcher wants to set alpha at .05, which is to say, have a 5% probability of rejecting the null hypothesis of no effect, if it is true.
>
> Using the fixed-sample stopping rule, the researcher would determine the number of subjects to be tested prior to performing the study. However, there is another type of stopping rule, called sequential stopping rules, which was first proposed by Wald in 1947. In a sequential stopping rule, the outcome ofthe statistical test could lead to testing more subjects. Thus, the number of subjects to be tested is not fixed in advance.[Fri98]
>
> According to Wainer (2000) an adaptive test can be considered complete after a predetermined number of items have been administered, when a predetermined level of measurement precision has been reached, or when a predetermined length of time has elapsed. The two most commonly used methods for determining when a computerized adaptive test is complete are the fixed length and variable length stopping rules.
>
> Under a fixed length stopping rule, an adaptive test is terminated when a predetermined number of items have been administered. Accordingly, all examinees are administered the same number of items, regardless of the degree of measurement precision achieved upon termination of the test. [...]
>
> In contrast, variable length stopping rules typically seek to achieve a certain degree of measurement precision for all examinees, even when doing so means that some examinees are given more items than others. Two types of variable length stopping rules have been used (Dodd, Koch, & De Ayala, 1993). These are the standard error (SE) stopping rule and the minimum information stopping rule. Of these, the most commonly used has been the SE stopping rule, which terminates an adaptive test when a predetermined standard error has been reached for the most recent examinee trait estimate (Boyd, Dodd, & Choi, 2010).[CGD10]

> One criticism of rules like O'Brien and Fleming is their rigidity in requiring a fixed maximum number of preplanned interim analyses. Greater flexibility is possible with the Peto-Haybittle rule, which simply specifies a fixed p value (often $p < 0.001$) for stopping early. For example, the European myocardial infarction amiodarone trial (EMIAT) is an ongoing study of 1500 patients at high risk after myocardial infarction comparing amiodarone (an antiarrhythmic drug) with placebo. Two year mortality is the primary end point, and the stopping guideline for efficacy is $p < 0.001$ in favour of amiodarone.[Poc92]

A clear stopping rule is necessary when the number of datapoints you could look at is practically infinite; we could survey every single person in the United States to determine their political views, but it is more feasible to pick a large enough sample of them to achieve a certain statistical accuracy. Alternatively, we could flip between data gathering and analysis until our target accuracy or statistical measure is reached, but this risks coming to a premature stop due to statistical fluctuations; had we just analyzed X more datapoints, our test statistic would switch from "statistically significant" to "not statistically significant" and our declaration of significance would be premature.

Dream has not completed a practically infinite number of random-seed any-percent Minecraft 1.16 runs, however. If our hypothesis is that Dream boosted his drop/barter rate since returning to that speedrun category, then the entire dataset consists of thirty-three runs where Blazes were killed for Blaze rods, and twenty-two runs where Ender pearls were bartered for, spread across six streaming sessions. There is no more data to be collected, and the data that is present is nowhere near large enough to pose an analytic challenge. If those datapoints were insufficient to establish the hypothesis, then a well-designed statistical test metric will conclude the evidence is inconclusive. If the fluctuations could be due to chance, then a well-designed metric will take that into account. The data cannot be cherry-picked to artificially boost the metric, as the MST report examines every applicable cherry.

The imposition of a binary decision threshold on continuous test statistics, and the resulting necessity of stopping rules, has led some researchers to call for the abandonment of statistical significance entirely.[MGG+19] Rather than have the researcher or publisher check if p-values or Bayes factors cross a certain threshold, they propose researchers and publishers take a holistic approach that also factors in "related prior evidence, plausibility of mechanism, study design and data quality, real world costs and benefits," and "novelty of finding" among other things.

As an outsider to the Minecraft speedrunning community, I do not have full knowledge of prior evidence. I do not have a good understanding of the costs or benefits of removing Dream's speed-running records, and should I make the wrong decision I will not pay the consequences. The best thing I can do is to instead lay out my full reasoning as clearly I can. If the community accepts the premises behind my arguements and can find few flaws in how I link them together, they are in a much better position to translate a continuous Bayes factor into a binary decision.

6.2 Lindley's Paradox

This analysis might seem like a waste: both the MST and PE reports conclude Dream cheated, and this report only avoids saying so because it is not the author's place to. If frequentist and Bayesian methods always come to the same conclusion, why does it matter which we choose?

Let's do a case study of Benex, another speedrunner the MST report gathered data on. Rather than repeat the prior chart, we'll do a simple frequentist analysis of Benex's data.

```
# Execute this cell to install all dependencies. Apologies for the spam.

# If you're running this on your own computer, I recommend altering "install mpmath"
#  to read "install --user mpmath", that way you don't need administrator access.
# You can get rid of the spam by changing "pip install" to "pip -q install", but on␣
↪Colab
#  you could miss a message that you need to restart your runtime. If you don't␣
↪restart,
#  the code won't work, hence why the default is to spam.
# A few error messages are fine, I usually get one about mismatched versions and the␣
↪code
```

(continues on next page)

(continued from previous page)

```
#  still runs.

!pip install mpmath myst_nb numpy pandas matplotlib scipy
```

```
Requirement already satisfied: mpmath in /usr/lib/python3/dist-packages (1.1.0)
```

```
Requirement already satisfied: myst_nb in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (0.10.2)
```

```
Requirement already satisfied: numpy in /usr/lib/python3/dist-packages (1.18.4)
```

```
Requirement already satisfied: pandas in /usr/lib/python3/dist-packages (1.0.5)
Requirement already satisfied: matplotlib in /usr/lib/python3/dist-packages (3.3.0)
Requirement already satisfied: scipy in /usr/lib/python3/dist-packages (1.5.2)
Requirement already satisfied: importlib-metadata in /usr/lib/python3/dist-packages␣
↪(from myst_nb) (1.6.0)
```

```
Requirement already satisfied: sphinx-togglebutton~=0.2.2 in /home/hjhornbeck/.local/
↪lib/python3.8/site-packages (from myst_nb) (0.2.3)
Requirement already satisfied: jupyter-sphinx==0.3.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (0.3.1)
```

```
Requirement already satisfied: jupyter-cache~=0.4.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (0.4.1)
```

```
Requirement already satisfied: ipython in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from myst_nb) (7.19.0)
```

```
Requirement already satisfied: docutils>=0.15 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from myst_nb) (0.16)
Requirement already satisfied: nbconvert~=5.6 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from myst_nb) (5.6.1)
```

```
Requirement already satisfied: nbformat~=5.0 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from myst_nb) (5.0.8)
```

```
Requirement already satisfied: ipywidgets<8,>=7.0.0 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (7.6.3)
```

```
Requirement already satisfied: sphinx<4,>=2 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from myst_nb) (3.4.3)
```

```
Requirement already satisfied: myst-parser~=0.12.9 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst_nb) (0.12.10)
```

```
Requirement already satisfied: pyyaml in /usr/lib/python3/dist-packages (from myst_
↪nb) (5.3.1)
```

```
Requirement already satisfied: setuptools in /usr/lib/python3/dist-packages (from␣
↪sphinx-togglebutton~=0.2.2->myst_nb) (49.3.1)
Requirement already satisfied: wheel in /usr/lib/python3/dist-packages (from sphinx-
↪togglebutton~=0.2.2->myst_nb) (0.34.2)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: attrs in /usr/lib/python3/dist-packages (from jupyter-
↪cache~=0.4.1->myst_nb) (19.3.0)
```

```
Requirement already satisfied: sqlalchemy~=1.3.12 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from jupyter-cache~=0.4.1->myst_nb) (1.3.22)
```

```
Requirement already satisfied: nbclient<0.6,>=0.2 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from jupyter-cache~=0.4.1->myst_nb) (0.5.1)
```

```
Requirement already satisfied: nbdime in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from jupyter-cache~=0.4.1->myst_nb) (2.1.0)
```

```
Requirement already satisfied: backcall in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (0.2.0)
Requirement already satisfied: pexpect>4.3; sys_platform != "win32" in /usr/lib/
↪python3/dist-packages (from ipython->myst_nb) (4.6.0)
Requirement already satisfied: traitlets>=4.2 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from ipython->myst_nb) (5.0.5)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/
↪lib/python3/dist-packages (from ipython->myst_nb) (3.0.6)
Requirement already satisfied: decorator in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (4.4.2)
```

```
Requirement already satisfied: jedi>=0.10 in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (0.17.0)
Requirement already satisfied: pygments in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from ipython->myst_nb) (2.7.3)
Requirement already satisfied: pickleshare in /usr/lib/python3/dist-packages (from␣
↪ipython->myst_nb) (0.7.5)
Requirement already satisfied: testpath in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (0.4.4)
Requirement already satisfied: jinja2>=2.4 in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (2.11.2)
Requirement already satisfied: bleach in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (3.2.1)
```

```
Requirement already satisfied: jupyter-core in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (4.6.3)
Requirement already satisfied: defusedxml in /usr/lib/python3/dist-packages (from␣
↪nbconvert~=5.6->myst_nb) (0.6.0)
Requirement already satisfied: entrypoints>=0.2.2 in /usr/lib/python3/dist-packages␣
↪(from nbconvert~=5.6->myst_nb) (0.3)
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/lib/python3/dist-packages␣
↪(from nbconvert~=5.6->myst_nb) (0.8.4)
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/lib/python3/dist-packages␣
↪(from nbconvert~=5.6->myst_nb) (1.4.2)
Requirement already satisfied: ipython-genutils in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from nbformat~=5.0->myst_nb) (0.2.0)
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/lib/python3/dist-
↪packages (from nbformat~=5.0->myst_nb) (3.2.0)
```

```
Requirement already satisfied: widgetsnbextension~=3.5.0 in /home/hjhornbeck/.local/
↪lib/python3.8/site-packages (from ipywidgets<8,>=7.0.0->myst_nb) (3.5.1)
Requirement already satisfied: ipykernel>=4.5.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from ipywidgets<8,>=7.0.0->myst_nb) (5.4.2)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >= "3.6" in /
↪home/hjhornbeck/.local/lib/python3.8/site-packages (from ipywidgets<8,>=7.0.0->myst_
↪nb) (1.0.0)
Requirement already satisfied: imagesize in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from sphinx<4,>=2->myst_nb) (1.2.0)
Requirement already satisfied: requests>=2.5.0 in /usr/lib/python3/dist-packages␣
↪(from sphinx<4,>=2->myst_nb) (2.23.0)
```

```
Requirement already satisfied: sphinxcontrib-jsmath in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.1)
Requirement already satisfied: sphinxcontrib-qthelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.3)
Requirement already satisfied: sphinxcontrib-serializinghtml in /home/hjhornbeck/.
↪local/lib/python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.1.4)
```

```
Requirement already satisfied: sphinxcontrib-htmlhelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.3)
Requirement already satisfied: sphinxcontrib-devhelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.2)
```

```
Requirement already satisfied: snowballstemmer>=1.1 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (2.0.0)
Requirement already satisfied: babel>=1.3 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from sphinx<4,>=2->myst_nb) (2.9.0)
Requirement already satisfied: packaging in /usr/lib/python3/dist-packages (from␣
↪sphinx<4,>=2->myst_nb) (20.4)
Requirement already satisfied: alabaster<0.8,>=0.7 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (0.7.12)
Requirement already satisfied: sphinxcontrib-applehelp in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from sphinx<4,>=2->myst_nb) (1.0.2)
```

```
Requirement already satisfied: markdown-it-py~=0.5.4 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from myst-parser~=0.12.9->myst_nb) (0.5.8)
```

```
Requirement already satisfied: jupyter-client>=6.1.5 in /usr/lib/python3/dist-
↪packages (from nbclient<0.6,>=0.2->jupyter-cache~=0.4.1->myst_nb) (6.1.6)
```

```
Requirement already satisfied: async-generator in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from nbclient<0.6,>=0.2->jupyter-cache~=0.4.1->myst_nb) (1.10)
Requirement already satisfied: nest-asyncio in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from nbclient<0.6,>=0.2->jupyter-cache~=0.4.1->myst_nb) (1.4.3)
```

```
Requirement already satisfied: colorama in /usr/lib/python3/dist-packages (from␣
↪nbdime->jupyter-cache~=0.4.1->myst_nb) (0.4.3)
Requirement already satisfied: GitPython!=2.1.4,!=2.1.5,!=2.1.6 in /home/hjhornbeck/.
↪local/lib/python3.8/site-packages (from nbdime->jupyter-cache~=0.4.1->myst_nb) (3.1.
↪12)
Requirement already satisfied: tornado in /usr/lib/python3/dist-packages (from nbdime-
↪>jupyter-cache~=0.4.1->myst_nb) (6.0.4)
Requirement already satisfied: six in /usr/lib/python3/dist-packages (from nbdime->
↪jupyter-cache~=0.4.1->myst_nb) (1.15.0)
Requirement already satisfied: notebook in /home/hjhornbeck/.local/lib/python3.8/site-
↪packages (from nbdime->jupyter-cache~=0.4.1->myst_nb) (6.1.6)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: pytz>=2015.7 in /usr/lib/python3/dist-packages (from␣
↪babel>=1.3->sphinx<4,>=2->myst_nb) (2020.1)
Requirement already satisfied: gitdb<5,>=4.0.1 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from GitPython!=2.1.4,!=2.1.5,!=2.1.6->nbdime->jupyter-cache~=0.4.
↪1->myst_nb) (4.0.5)
```

```
Requirement already satisfied: prometheus-client in /usr/lib/python3/dist-packages␣
↪(from notebook->nbdime->jupyter-cache~=0.4.1->myst_nb) (0.7.1)
Requirement already satisfied: argon2-cffi in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from notebook->nbdime->jupyter-cache~=0.4.1->myst_nb) (20.1.0)
```

```
Requirement already satisfied: Send2Trash in /usr/lib/python3/dist-packages (from␣
↪notebook->nbdime->jupyter-cache~=0.4.1->myst_nb) (1.5.0)
Requirement already satisfied: terminado>=0.8.3 in /home/hjhornbeck/.local/lib/
↪python3.8/site-packages (from notebook->nbdime->jupyter-cache~=0.4.1->myst_nb) (0.9.
↪2)
```

```
Requirement already satisfied: pyzmq>=17 in /usr/lib/python3/dist-packages (from␣
↪notebook->nbdime->jupyter-cache~=0.4.1->myst_nb) (19.0.2)
Requirement already satisfied: smmap<4,>=3.0.1 in /home/hjhornbeck/.local/lib/python3.
↪8/site-packages (from gitdb<5,>=4.0.1->GitPython!=2.1.4,!=2.1.5,!=2.1.6->nbdime->
↪jupyter-cache~=0.4.1->myst_nb) (3.0.4)
Requirement already satisfied: cffi>=1.0.0 in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from argon2-cffi->notebook->nbdime->jupyter-cache~=0.4.1->myst_nb)␣
↪(1.14.4)
```

```
Requirement already satisfied: ptyprocess; os_name != "nt" in /home/hjhornbeck/.local/
↪lib/python3.8/site-packages (from terminado>=0.8.3->notebook->nbdime->jupyter-cache~
↪=0.4.1->myst_nb) (0.7.0)
Requirement already satisfied: pycparser in /home/hjhornbeck/.local/lib/python3.8/
↪site-packages (from cffi>=1.0.0->argon2-cffi->notebook->nbdime->jupyter-cache~=0.4.
↪1->myst_nb) (2.20)
```

```
from fractions import Fraction

from math import log,factorial
import matplotlib.pyplot as plt
from mpmath import mp
from myst_nb import glue

import numpy as np

import pandas as pd
from scipy.optimize import differential_evolution
from scipy.stats import beta,binom,nbinom


dpi         = 200  # change this to increase/decrease the resolution charts are made␣
↪at
book_output = True ### changing this to False will allow you to view the plots in a␣
↪Jupyter notebook

def fig_show( fig, name ):
```

(continues on next page)

(continued from previous page)

```
    """A helper to control how we're displaying figures."""
    global book_output

    if book_output:
        glue( name, fig, display=False )
        plt.close();
    else:
        fig.show()
```

```
# missing "simple_bernoulli_bf.py"? Uncomment this line to grab it from the repository
# !wget -c "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_cheating/main/
↪simple_bernoulli_bf.py"

from simple_bernoulli_bf import prior, posterior_H_fair, BF_H_fair_H_cheat
```

```
# missing the data files? Uncommenting and running this cell might retrieve them

# !mkdir data
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/blaze.benex.tsv"
# !wget -cP data "https://raw.githubusercontent.com/hjhornbeck/bayes_speedrun_
↪cheating/main/data/bartering.benex.tsv"
```

```
colours = ['b', 'g', 'r', 'c', 'm', 'y', 'k']  # makes colouring some of the␣
↪following easier
colours = colours + colours                    # double-up to allow for extra data

blaze_players = ['benex']
pearl_players = blaze_players

blaze_rods = {p:pd.read_csv(f'data/blaze.{p}.tsv',sep="\t") for p in blaze_players}
bartering  = {p:pd.read_csv(f'data/bartering.{p}.tsv',sep="\t") for p in pearl_
↪players}

blaze_rods['benex'].set_index('n').transpose()
```

n	11	15	14	8	13	12	16	14	7	14	11
k	6	7	6	7	6	7	7	10	3	5	8

```
fig, (ax1,ax2) = plt.subplots(nrows=1, ncols=2, figsize=(8, 4), dpi=dpi, facecolor='w
↪', edgecolor='k')

i = 0       # allow for copy-paste code
p = 'benex'


# Blaze rods first
r_fair = Fraction(1,2)
priors = [prior( r_fair, scale ) for scale in [Fraction(4,3), 4, 12]]

count = len(blaze_rods[p])
x     = np.arange(1, count+1)
sum_n = np.cumsum( blaze_rods[p]['n'] )
sum_k = np.cumsum( blaze_rods[p]['k'] )
```

(continues on next page)

(continued from previous page)

```
# p-values
y = [1 - binom.cdf( sum_k[i], sum_n[i], r_blaze ) for i in range(count)]
ax1.plot( x, y, '-r', label="blaze rods" )

# my Bayes factor
y = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[1]) for i in range(count)]
ax2.plot( x, y, '-r' )

y_low = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[0]) for i in␣
↪range(count)]
y_hig = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[2]) for i in␣
↪range(count)]
ax2.fill_between( x, [float(v) for v in y_low], [float(v) for v in y_hig], \
                     alpha=0.2, color='r' )


# Ender pearls
r_fair = Fraction(20, 423)
priors = [prior( r_fair, scale ) for scale in [Fraction(4,3), 4, 12]]

count = len(bartering[p])
x     = np.arange(1, count+1)
sum_n = np.cumsum( bartering[p]['n'] )
sum_k = np.cumsum( bartering[p]['k'] )

# p-values
y = [1 - binom.cdf( sum_k[i], sum_n[i], r_pearl ) for i in range(count)]
ax1.plot( x, y, '-b' )
glue( 'benex_pval_12', y[11], display=False )

# my Bayes factor
y = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[1]) for i in range(count)]
ax2.plot( x, y, '-b', label="pearl barters" )
glue( 'benex_bf_12', float(y[11]), display=False )

y_low = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[0]) for i in␣
↪range(count)]
y_hig = [BF_H_fair_H_cheat(sum_k[i], sum_n[i], r_fair, *priors[2]) for i in␣
↪range(count)]
ax2.fill_between( x, [float(v) for v in y_low], [float(v) for v in y_hig], \
                     alpha=0.2, color='b' )


# add the trimmings to both charts
x = range( 1, max(len(blaze_rods[p]), len(bartering[p])) )
ax1.plot( x, [0.05 for v in x], '--k', label='p = 0.05' )
ax1.plot( x, [0.001 for v in x], '--k', label='p = 0.001' )

ax2.plot( x, [19 for v in x], '--k', label='19:1' )
ax2.plot( x, [1 for v in x], '--k', label='break even' )


ax1.set_xlabel("run")
ax1.set_xticks([1,len(blaze_rods['benex']),len(bartering['benex'])])
ax1.set_ylabel('p-value')
ax1.set_yscale("log")
```

(continues on next page)

(continued from previous page)

```
ax2.set_xlabel("run")
ax2.set_xticks([1,len(blaze_rods['benex']),len(bartering['benex'])])
ax2.set_ylabel('H_fair / H_cheat')
ax2.set_yscale("log")

ax1.legend()
ax2.legend()

fig_show( fig, 'fig:benex_both' )
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-b85abe410851> in <module>
     15
     16 # p-values
---> 17 y = [1 - binom.cdf( sum_k[i], sum_n[i], r_blaze ) for i in range(count)]
     18 ax1.plot( x, y, '-r', label="blaze rods" )
     19

<ipython-input-6-b85abe410851> in <listcomp>(.0)
     15
     16 # p-values
---> 17 y = [1 - binom.cdf( sum_k[i], sum_n[i], r_blaze ) for i in range(count)]
     18 ax1.plot( x, y, '-r', label="blaze rods" )
     19

NameError: name 'r_blaze' is not defined
```

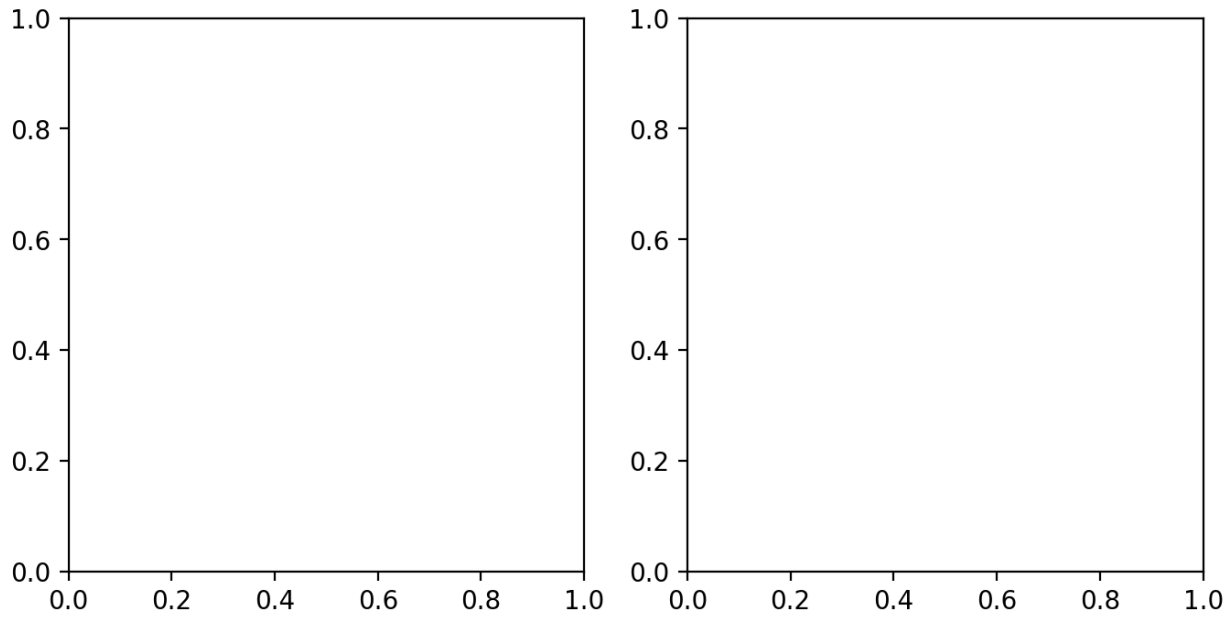

Fig. 6.1: A comparison of p-values and my Bayes factor for Benex's record on Blaze rod drops and Ender pearl barters. Both metrics are calculated according to the cumulative n and k as of that run.

Some Cold Water

I should point out that this imagined world is not our own. The MST team applied corrections for multiple comparisons that would likely have watered down the p-value. Benex's first five runs were part of one stream, while his next three were in a second stream, so the run that crossed the line was at the start of a new stream and followed by two runs that watered down his p-values. It's highly unlikely Benex switched their Ender pearl barter rate mid-stream, so that sixth run can be safely declared an outlier and discarded.

These graphs tell two very different stories. We could imagine a universe where the Minecraft Speedrunning Team invoking the Peto-Haybittle stopping rule, as it is well regarded though considered too conservative to some researchers[SG05]. When Benex's p-value crossed the $p < 0.001$ threshold on his sixth run, they declare him to have cheated and strip him of any records he held.

We could also imagine a universe where the MST invoked my Bayes prior, observed the exact same run from Benex, noted his odds of fair play never dipped below 6:1 for my choice of prior, and did nothing. At no point would this universe's MST believe Benex tweaked his Ender pearl barter rate.

This is just barely an example of Lindley's Paradox.

> An example is produced to show that, if H is a simple hypothesis and x the result of an experiment, the following two phenomena can occur simultaneously:
>
> (i) a significance test for H reveals that x is significant at, say, the 5% level;(ii) the posterior probability of H, given x, is, for quite small prior probabilities of H, as high as 95%.
>
> Clearly the common-sense interpretations of (i) and (ii) are in direct conflict. The phenomenon is fairly general with significance tests and casts doubts on the meaning of a significance level in some circumstanices.[Lin57]

A Neat Coincidence

As luck would have it, that was also the last run he made of his third stream. The Minecraft Speedrunning Team only examined six of his streams, so that twelfth run was a natural decision point for a stopping rule.

On the twelfth run, Benex's cumulative p-value is , while my Bayes factor is about :1, which when translated to a percentage is over the 95% line. Dream's data may be strong enough to point in one direction, but the data from other speedrunners is not guaranteed to do the same. Indeed, anyone interested in cheating would likely decrease their advantage below what was listed in the PE report, increasing the odds of achieving Lindley's paradox.

The debate between frequentist and Bayesian statistics can have a real-world consequences, and is not to be dismissed lightly.

CHAPTER

SEVEN

FUTURE WORK

While I cannot provide a solid yes or no answer to "did Dream cheat," I have provided several other things which may prove useful. My analysis may be applicable to any Bernoulli process where we do not know the true rate. If we have a hypothesis that the rate is a specific value, and no strong alternative hypothesis, then the two Python routines `posterior_H_fair()` and `BF_H_fair_H_cheat()` can be invoked to generate a Bayes factor for reasonable mathematical definitions of those hypotheses. Both routines run quite fast; most of this notebook was developed on an Asus Transformer, and the slowest cell takes about thirty seconds to execute. Both routines are also quite flexible, as they allow arbitrary Beta priors to be supplied and have arbitrary precision so that math overflow isn't a problem. This makes those routines applicable well beyond the simple question of "did Dream cheat?"

I have also tried to make it as easy as possible to build another analysis on top of this one. If, for instance, you do not agree with my definition of H_{cheat}, you can develop your own mathematical version and use that to weight the posterior. Just as I tried to take the best parts of the MST and PE reports to build a new analysis, you could do the same with this report.

For instance, one idea not explored here is heirarchical models. The PE report attempted to build such a model for Ender pearl barters using Monte Carlo simulations, but a better approach is to invoke Stan or emcee to build and sample from the posterior. This would allow you to test the success rate of barters, the number of Ender pearls dropped per successful barter, and the amount of gold a speedrunner could gather simultaneously. Rather than assert whether the speedrunner's goal was ten or twelve Ender pearls, as the PE report's simulation does, they can be free parameters in the model that the MCMC engine will explore for you.

At minimum, I at least hope to have provided a different and more rigorous approach towards analyzing whether or not a speedrunner modified their game.

CHAPTER

EIGHT

BIBLIOGRAPHY

BIBLIOGRAPHY

[CGD10] Seung W. Choi, Matthew W. Grady, and Barbara G. Dodd. A New Stopping Rule for Computerized Adaptive Testing. *Educational and psychological measurement*, 70(6):1–17, December 2010. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3028267/, doi:10.1177/0013164410387338.

[CW09] Matthew P.A. Clark and Brian D. Westerberg. How random is the toss of a coin? *CMAJ*, 181(12):E306–E308, 2009. URL: https://www.cmaj.ca/content/181/12/E306, arXiv:https://www.cmaj.ca/content/181/12/E306.full.pdf, doi:10.1503/cmaj.091733.

[Coh94] Jacob Cohen. The earth is round (p \textless .05). *American Psychologist*, 49(12):997–1003, 1994. doi:10.1037/0003-066X.49.12.997.

[Col14] David Colquhoun. An investigation of the false discovery rate and the misinterpretation of p-values. *Royal Society open science*, 1(3):140216, 2014.

[Cum08] Geoff Cumming. Replication and p intervals: p values predict the future only vaguely, but confidence intervals do much better. *Perspectives on Psychological Science*, 3(4):286–300, 2008. PMID: 26158948. URL: https://doi.org/10.1111/j.1745-6924.2008.00079.x, arXiv:https://doi.org/10.1111/j.1745-6924.2008.00079.x, doi:10.1111/j.1745-6924.2008.00079.x.

[F+34] Ronald Aylmer Fisher and others. Statistical methods for research workers. *Statistical methods for research workers.*, 1934.

[Fri98] Robert W. Frick. A better stopping rule for conventional statistical tests. *Behavior Research Methods, Instruments, & Computers*, 30(4):690–697, December 1998. URL: https://doi.org/10.3758/BF03209488, doi:10.3758/BF03209488.

[Gig93] Gerd Gigerenzer. The superego, the ego, and the id in statistical reasoning. *A handbook for data analysis in the behavioral sciences: Methodological issues*, pages 311–339, 1993.

[Goo99] Steven Goodman. Toward evidence-based medical statistics. 1: the p value fallacy. *Annals of Internal Medicine*, 130(12):995–1004, 1999. PMID: 10383371. URL: https://www.acpjournals.org/doi/abs/10.7326/0003-4819-130-12-199906150-00008, arXiv:https://www.acpjournals.org/doi/pdf/10.7326/0003-4819-130-12-199906150-00008, doi:10.7326/0003-4819-130-12-199906150-00008.

[Goo08] Steven Goodman. A dirty dozen: twelve p-value misconceptions. *Seminars in Hematology*, 45(3):135 – 140, 2008. Interpretation of Quantitative Research. URL: http://www.sciencedirect.com/science/article/pii/S0037196308000620, doi:https://doi.org/10.1053/j.seminhematol.2008.04.003.

[Gri12] David Griffiths. *Statistics in Gambling*, chapter, pages 1–4. American Cancer Society, 2012. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471667196.ess0845.pub3, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471667196.ess0845.pub3, doi:https://doi.org/10.1002/0471667196.ess0845.pub3.

[HCEVD15] Lewis G. Halsey, Douglas Curran-Everett, Sarah L. Vowler, and Gordon B. Drummond. The fickle P value generates irreproducible results. *Nature Methods*, 12(3):179–185, March 2015. URL: https://www.nature.com/articles/nmeth.3288/, doi:10.1038/nmeth.3288.

[HL08] Raymond Hubbard and R. Murray Lindsay. Why P Values Are Not a Useful Measure of Evidence in Statistical Significance Testing. *Theory & Psychology*, 18(1):69–88, February 2008. Publisher: SAGE Publications Ltd. URL: https://doi.org/10.1177/0959354307086923, doi:10.1177/0959354307086923.

[Joh13] Valen E. Johnson. Revised standards for statistical evidence. *Proceedings of the National Academy of Sciences*, 110(48):19313–19317, 2013. URL: https://www.pnas.org/content/110/48/19313, arXiv:https://www.pnas.org/content/110/48/19313.full.pdf, doi:10.1073/pnas.1313476110.

[KR95] Robert E. Kass and Adrian E. Raftery. Bayes factors. *Journal of the American Statistical Association*, 90(430):773–795, 1995. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1995.10476572, arXiv:https://www.tandfonline.com/doi/pdf/10.1080/01621459.1995.10476572, doi:10.1080/01621459.1995.10476572.

[Lin57] D. V. Lindley. A Statistical Paradox. *Biometrika*, 44(1/2):187–192, 1957. URL: https://www.jstor.org/stable/2333251, doi:10.2307/2333251.

[Lyo13] Louis Lyons. Discovering the Significance of 5 sigma. *arXiv:1310.1284 [hep-ex, physics:hep-ph, physics:physics]*, October 2013. arXiv: 1310.1284. URL: http://arxiv.org/abs/1310.1284.

[MGG+19] Blakeley B. McShane, David Gal, Andrew Gelman, Christian Robert, and Jennifer L. Tackett. Abandon statistical significance. *The American Statistician*, 73(sup1):235–245, 2019. URL: https://doi.org/10.1080/00031305.2018.1527253, arXiv:https://doi.org/10.1080/00031305.2018.1527253, doi:10.1080/00031305.2018.1527253.

[MHR+16] Richard D Morey, Rink Hoekstra, Jeffrey N Rouder, Michael D Lee, and Eric-Jan Wagenmakers. The fallacy of placing confidence in confidence intervals. *Psychonomic bulletin & review*, 23(1):103–123, 2016.

[Pil91] David B. Pillemer. One- versus two-tailed hypothesis tests in contemporary educational research. *Educational Researcher*, 20(9):13–17, 1991. URL: https://doi.org/10.3102/0013189X020009013, arXiv:https://doi.org/10.3102/0013189X020009013, doi:10.3102/0013189X020009013.

[Poc92] S. J. Pocock. When to stop a clinical trial. *BMJ : British Medical Journal*, 305(6847):235–240, July 1992. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1882707/.

[SG05] Kenneth F. Schulz and David A. Grimes. Multiplicity in randomised trials II: subgroup and interim analyses. *The Lancet*, 365(9471):1657–1661, May 2005. URL: https://www.thelancet.com/journals/lancet/article/PIIS0140-6736(05)66516-6/abstract, doi:10.1016/S0140-6736(05)66516-6.

[ST03] John D. Storey and Robert Tibshirani. Statistical significance for genomewide studies. *Proceedings of the National Academy of Sciences*, 100(16):9440–9445, 2003. URL: https://www.pnas.org/content/100/16/9440, arXiv:https://www.pnas.org/content/100/16/9440.full.pdf, doi:10.1073/pnas.1530509100.

[Syv98] Anne Randi Syversveen. Noninformative bayesian priors. interpretation and problems with construction and applications. *preprint*, 1998.

[Van14] Jake VanderPlas. Frequentism and Bayesianism: A Python-driven Primer. *arXiv:1411.5018 [astro-ph]*, November 2014. 00000 arXiv: 1411.5018. URL: http://arxiv.org/abs/1411.5018.