Data Simulation

Dr Andrew J. Stewart

E: drandrewjstewart@gmail.com
T: @ajstewart_lang
G: ajstewartlang

Data Simulation – Why?

Data simulation allows you to do a number of things:

- Determine whether your design supports the kinds of analyses you are planning to do (especially important if you pre-registered your analysis plan).

- Determine whether the sample size and number of observations you are collecting is sufficient for you to detect the magnitude of the effect you are predicting with a reasonable level of precision.

- Write your analysis script before you’ve even collected your data thus making everything more efficient.

The `rnorm()` function

The `rnorm()` function allows us to sample n times from the normal distribution where we can specify both the mean and the standard deviation of the distribution we want to sample from. The function takes three parameters - the number of samples, the mean and the standard deviation of the distribution to sample from.

```
> rnorm(5, 0, 1)
[1] 0.24751016    0.32518029 1.12242126    2.13538261    -0.04670306

> rnorm(5, 0, 1)
[1] 0.1661151     0.1937463  -0.7434664    1.0375703     2.2625231
```

Notice that the two times we call the `rnorm()` function we get different random samples...

Use `set.seed()` to ensure reproducibility

We want to make sure we can replicate our sample - we can use the `set.seed()` function to specify the seed of the randomisation (so we can rerun the code and get the same result).

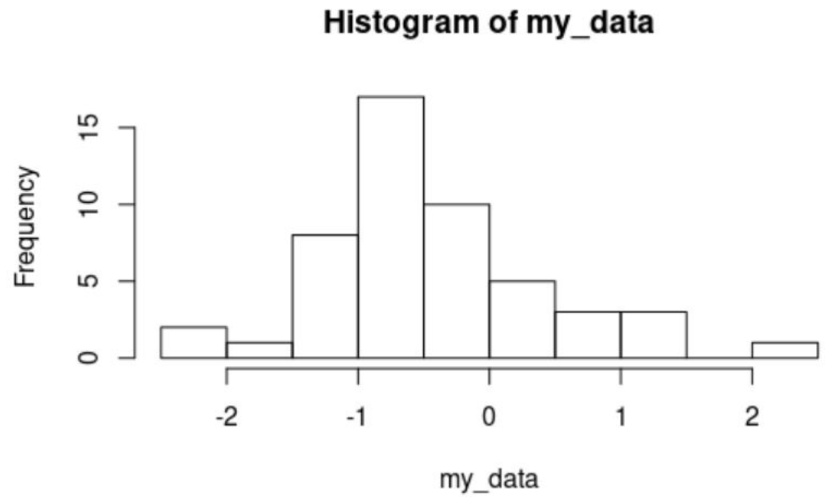
```
> set.seed(1234)
> rnorm(5, 0, 1)
[1] -1.2070657   0.2774292 1.0844412 -2.3456977 0.4291247

> set.seed(1234)
> rnorm(5, 0, 1)
[1] -1.2070657   0.2774292 1.0844412 -2.3456977 0.4291247
```

Now the two samples are identical.

Plotting our simulated data

We can map our random sample onto a new variable I'm calling `my_data` and then plot a histogram of the values. Here is a N = 50 sample.

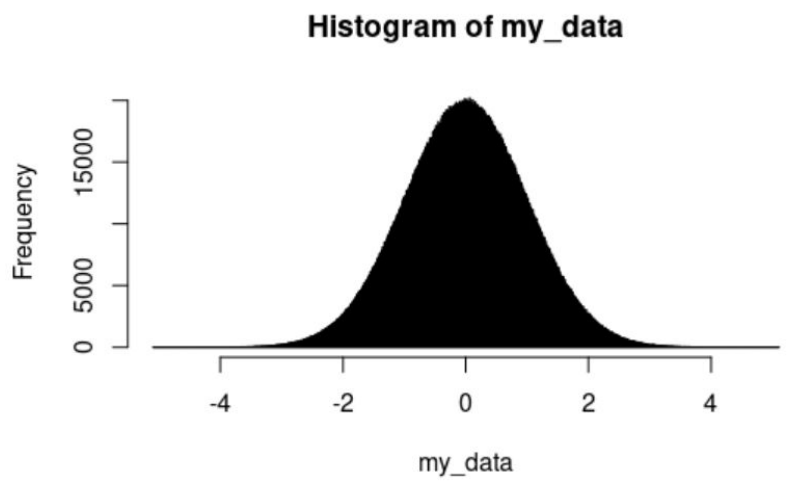
```
> set.seed(1234)
> my_data <- rnorm(50, 0, 1)
> hist(my_data)
```



Increasing our sample size

The larger the sample the more representative it is of the population it is drawn from. Here is a N = 5 million sample.

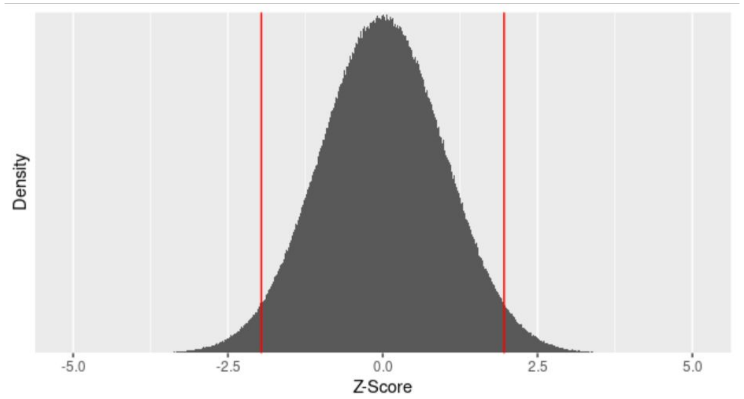
```
> set.seed(1234)
> my_data <- rnorm(5000000, 0, 1)
> hist(my_data, breaks = 1000)
```



What % of data points are more than 1.96 sds either side of the mean?

```
my_data %>%
  as_tibble() %>%
  filter(value > 1.96 | value < -1.96) %>%
  count()

# A tibble: 1 x 1
      n
   <int>
1 250434

> (250434/5000000) * 100

> [1] 5.00868
```

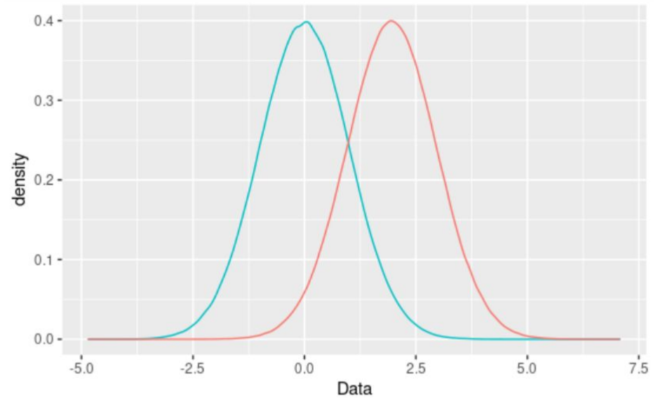
We calculate that 5% of data points are more than 1.96 sds from the mean (which is what we'd expect given we've just plotted the standard normal distribution)...



Plotting simulated data from two distributions

Simulating data sampled from 2 distributions and plotting them on the same graph:

```
set.seed(1234)
condition1 <- rnorm(1000000, 0, 1)
condition2 <- rnorm(1000000, 1.96, 1)
my_data <- as_tibble(cbind(condition1,
condition2))

ggplot(my_data) +
  geom_density(aes(x = condition1,
             y = ..density.., colour = "red")) +
  geom_density(aes(x = condition2,
             y = ..density.., colour = "green")) +
  xlab("Data") +
  guides(colour = FALSE)
```



Useful functions for building simulated data sets

Previously we have used a function `c()` which combines elements into one vector.

On the previous slide you might have spotted the function `cbind()` which combines vectors by column.

```
> a <- c(1, 2, 3)
> b <- c(4, 5, 6)
> cbind(a,b)

a b
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

Useful functions for building simulated data sets

Related to `cbind()` there is `rbind()` which combines vectors by row:

```
> a <- c(1, 2, 3)
> b <- c(4, 5, 6)
> rbind(a, b)

   [,1]  [,2]  [,3]
a  1     2     3
b  4     5     6
```

Useful functions for building simulated data sets

There are a few other functions we'll use when we simulate data - there include `seq()` and `rep()`.

`seq()` will generate a sequence from one number to another - you can also specify the number to increment the sequence by.

You can map this onto a new variable...

```
> seq(from = 1, to = 10, by = 1)
[1] 1 2 3 4 5 6 7 8 9 10
> seq(from = 1, to = 10, by = 2)
[1] 1 3 5 7 9
```

Useful functions for building simulated data sets

`rep()` stands for replicate and allows you to replicate elements in a vector a certain number of times:

```
> rep(1:5, times = 2)
[1] 1 2 3 4 5 1 2 3 4 5
```

You can also embed one function within another:

```
> rep(seq(from = 1, to = 10, by = 2), times = 2)
[1] 1 3 5 7 9 1 3 5 7 9
```

Useful functions for building simulated data sets

The vector you’re replicating doesn’t have to just be numbers:

```
> rep("fast", times = 12)
[1] "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast"
```

and you can use the combine function `c()` within the `rep()` function:

```
> c(rep("fast", times = 12), rep("slow", times = 12))
[1] "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast"
[13] "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow"
```

Simulating a data set

We can start using what we know so far to simulate a data set. Let's simulate data from an independent samples experiment with one factor with two levels.

Each of the 24 participants will have a measure - participants 1-12 are in the ‘fast’ condition and 13-24 in the ‘slow’ condition.

First let's create a vector for our participant ID number. It will range from 1 to 24

```
> participant <- seq(1:24)
> participant
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

Simulating a data set

Now we need to create the conditions - Condition 1 we will label “fast” and Condition 2 we will label “slow”.

We use the `c()` function to combine the arguments that follow it (i.e., "fast" and "slow") into a vector.

```
> condition <- c(rep("fast", times = 12), rep("slow", times = 12))
> condition
[1] "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast" "fast"
[13] "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow"
```

Simulating a data set

Now we need to simulate our data - we will sample from the normal distribution so will use the `rnorm()` function.

We want to simulate the data for our "fast" condition as coming from a distribution with a mean = 1000 and sd = 50, and the data for our "slow" condition from a distribution with a mean = 1020 and sd = 50.

We need to make sure we set up the order of our `rnorm()` function in the same way as we did for specifying the condition variable (i.e., sampling 12 times for the “fast” condition and then 12 for the “slow”).

Simulating a data set

To make sure we can reproduce these random samples in future, we can use the function `set.seed()` to specify the start of the random number generation.

```
> set.seed(1234)
> dv <- c(rnorm(12, 1000, 50), rnorm(12, 1020, 50))
> dv
[1] 939.6467 1013.8715 1054.2221 882.7151 1021.4562 1025.3028 971.2630 972.6684 971.7774
[10] 955.4981 976.1404 950.0807 981.1873 1023.2229 1067.9747 1014.4857 994.4495 974.4402
[19] 978.1414 1140.7918 1026.7044 995.4657 997.9726 1042.9795
```

Simulating a data set

We now need to combine our 3 columns (participant, condition, dv) into a tibble. We use the `cbind()` function to first bind the three variables together as columns, and then `as_tibble()` to convert these three combined columns to a tibble that I'm mapping onto the variable `my_data`.

```
> my_data <- as_tibble(cbind(participant, condition, dv))
> my_data
# A tibble: 24 x 3
participant condition dv
<chr> <chr> <chr>
1 1   fast  939.646712530729
2 2   fast  1013.87146210553
3 3   fast  1054.22205883415
4 4   fast  882.715114868533
5 5   fast  1021.45623444055
6 6   fast  1025.30279460788
7 7   fast  971.263001993268
8 8   fast  972.668407210791
9 9   fast  971.777400045336
10 10 fast  955.498108547795
# ... with 14 more rows
```

Simulating a data set

As our three columns are all listed as character type, we need to change `condition` to a factor and `dv` to an integer.

```
> my_tidied_data <- my_data %>%
                    mutate(condition = factor(condition), dv = as.integer(dv))
> my_tidied_data
# A tibble: 24 x 3
   participant condition   dv
   <chr>         <fct>      <int>
 1 1             fast       939
 2 2             fast       1013
 3 3             fast       1054
 4 4             fast       882
 5 5             fast       1021
 6 6             fast       1025
 7 7             fast       971
 8 8             fast       972
 9 9             fast       971
10 10            fast       955
# … with 14 more rows
```

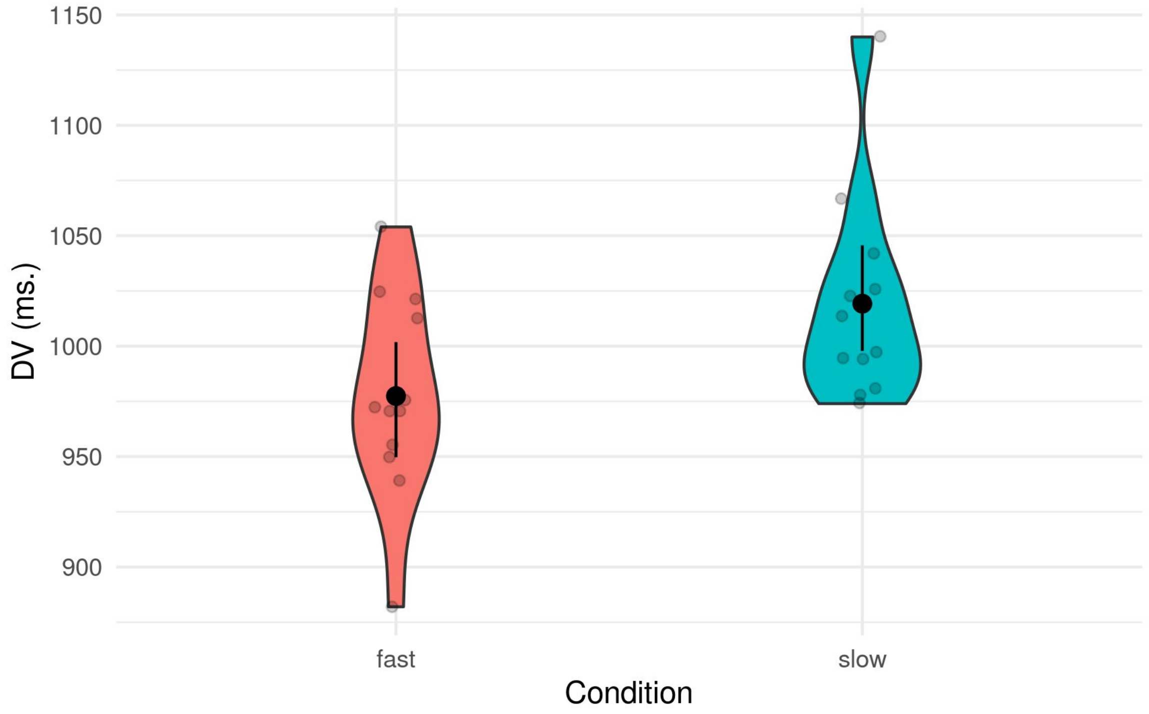
Does our simulated data look as expected?

So the tibble structure looks like what we expect, but do the data look like what we expect?

Remember, we sampled the ‘fast’ group from a distribution with a mean of 1000, and the ‘slow’ group from a distribution with a mean of 1020 - both with a standard deviation of 50.

```
ggplot(my_tidied_data, aes(x = condition, y = dv, fill = condition)) +
  geom_violin(width = .25) +
  stat_summary(fun.data = "mean_cl_boot", colour = "black") +
  geom_jitter(alpha = .2, width = .05) +
  guides(fill = FALSE) +
  labs(x = "Condition", y = "DV (ms.)") +
  theme_minimal()
```

Does our simulated data look as expected?



Does our simulated data look as expected?

```
my_tidied_data %>%
  group_by(condition) %>%
  summarise(mean_dv = mean(dv), sd_dv = sd(dv))

# A tibble: 2 x 3
  condition    mean_dv     sd_dv
  <fct>        <dbl>       <dbl>
1 fast         977.        46.0
2 slow         1019.       47.1
```

Looks pretty much like what we'd expect given a bit of sampling error...

Do we have a significant difference?

We can now perform an independent samples t-test to see if the conditions differ:

```
> t.test(filter(my_tidied_data, condition == "fast")$dv, filter(my_tidied_data, condition ==
"slow")$dv, paired = FALSE)

Welch Two Sample t-test
data: filter(my_tidied_data, condition == "fast")$dv and filter(my_tidied_data, condition
=="slow")$dv
t = -2.202, df = 21.987, p-value = 0.03845
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
     -81.233786 -2.432881
sample estimates:
mean of x mean of y
977.4167 1019.2500
```

The important stuff is a bit buried in the text - wouldn’t it be great if we could somehow extract it and save it?

We can save the result of this t-test using the `broom::tidy()` function. This converts the output of the t-test into a tibble.

```
result <- tidy(t.test(filter(my_tidied_data, condition == "fast")$dv,
                  filter(my_tidied_data, condition == "slow")$dv, paired = FALSE))
result

# A tibble: 1 x 10
  estimate estimate1 estimate2 statistic p.value parameter conf.low conf.high method
     <dbl>     <dbl>     <dbl>     <dbl>   <dbl> <dbl>     <dbl>     <dbl> <chr>
1    -41.8     977.      1019.     -2.20  0.0385      22.0 -81.2     -2.43 Welch…
# … with 1 more variable: alternative <chr>
```

We can reference columns in this tibble - for example, just to get the *p*-value we can type:

```
> result$p.value
[1] 0.03845285
```

Wouldn’t it be amazing if we could run t-tests on 100 simulations and save the result of each in a tibble that ends up containing the *p*-values for all the results?

Loops

Loops involve repeating a command or a set of commands surrounded by {} a certain number of times - popular with kids in the 1980s...unpopular with many “serious” R programmers because they can be inefficient...but liked by me because they are relatively easy to understand.

```
for (i in 1:10) {
    print(i)
}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

We can use a loop to index a column in a tibble - here is the tibble `my_tidied_data` again:

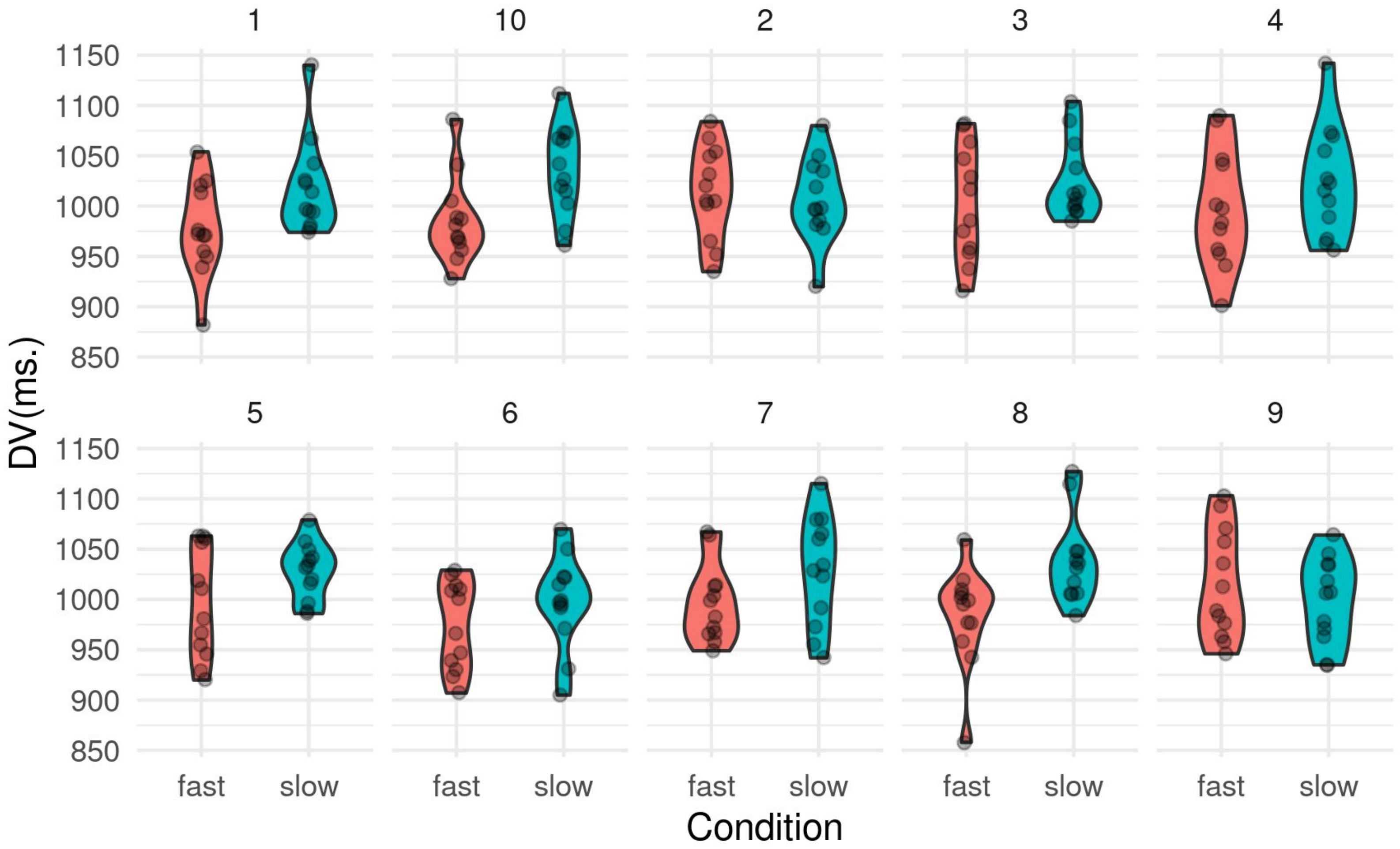
```
> my_tidied_data
# A tibble: 24 x 3
   participant   condition    dv
   <chr>         <fct>        <int>
 1 1             fast         939
 2 2             fast         1013
 3 3             fast         1054
 4 4             fast         882
 5 5             fast         1021
 6 6             fast         1025
 7 7             fast         971
 8 8             fast         972
 9 9             fast         971
10 10            fast         955
# … with 14 more rows
```

And here we use i to index the column dv in the `my_tidied_data` tibble:

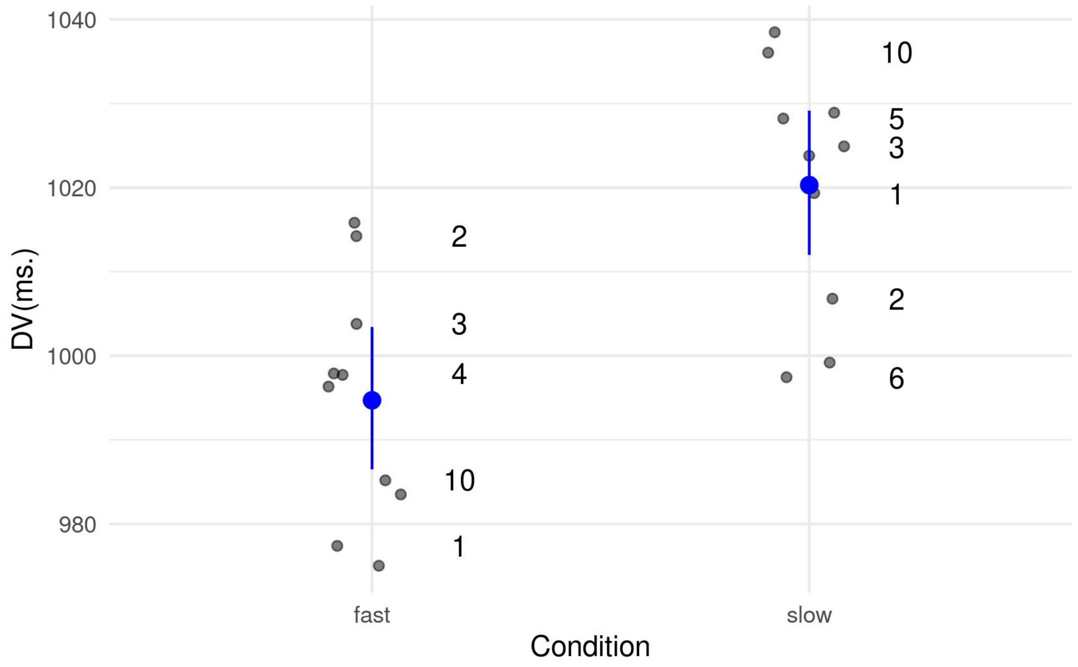
```
for (i in 1:10) {
  print(my_tidied_data$dv[i])
}

[1] 939
[1] 1013
[1] 1054
[1] 882
[1] 1021
[1] 1025
[1] 971
[1] 972
[1] 971
[1] 955
```

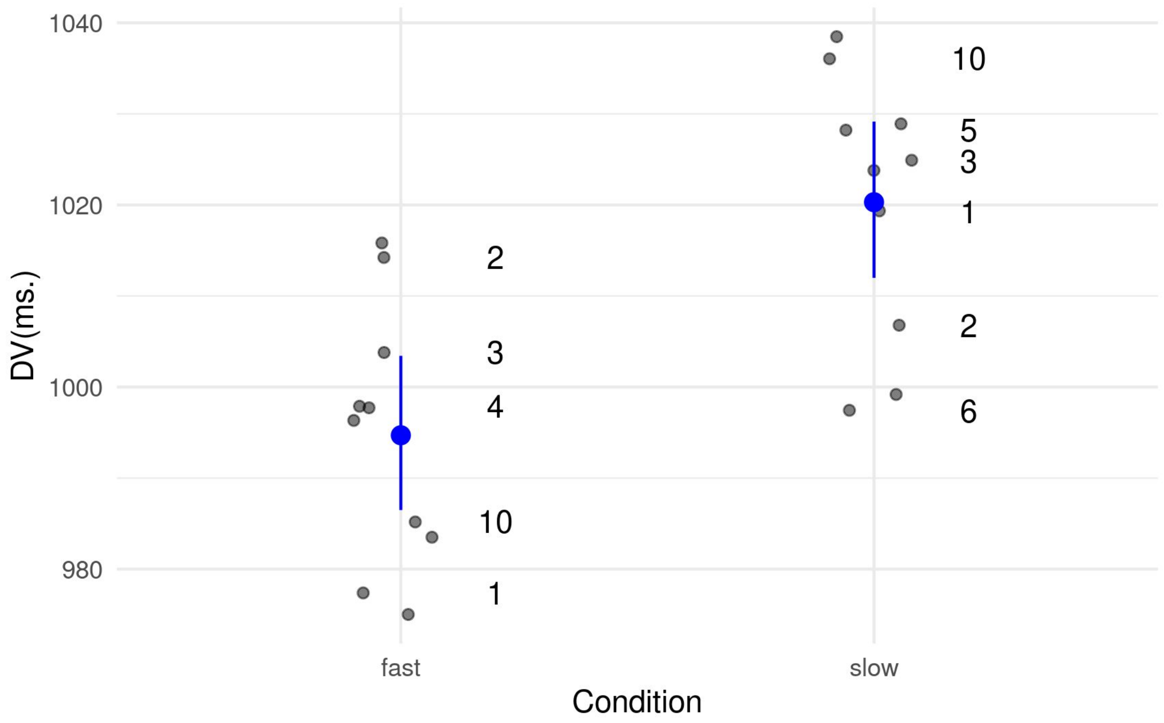
We can put together all we know so far to simulate data from 10 experiments (that I'm referring to as samples):

```r
total_samples <- 10
sample_size <- 24
participant <- rep(1:sample_size)
condition <- c(rep("fast", times = sample_size/2),
               rep("slow", times = sample_size/2))
all_data <- NULL

for (i in 1:total_samples) {
  sample <- i
  set.seed(1233 + i)
  dv <- c(rnorm(sample_size/2, 1000, 50), rnorm(sample_size/2, 1020, 50))
  my_data <- as_tibble(cbind(participant, condition, dv, sample))
  all_data <- rbind(my_data, all_data)
}

all_tidied_data <- all_data %>%
  mutate(condition = factor(condition), dv = as.integer(dv))
```


1
10
2
3
4
5
6
7
8
9
1150
1100
1050
1000
950
900
850
DV(ms.)
fast
slow
Condition

```
all_tidied_data %>%
  group_by(condition, sample) %>%
  summarise(average = mean(dv)) %>%
  ggplot(aes(x = condition, y = average, group = condition, label = sample)) +
  geom_jitter(width = .1, alpha = .5) +
  stat_summary(fun.data = "mean_cl_boot", colour = "blue") +
  geom_text(check_overlap = TRUE, nudge_x = .2, nudge_y = 0, colour = "black") +
  labs(x = "Condition", y = "DV(ms.)") +
  theme_minimal()
```



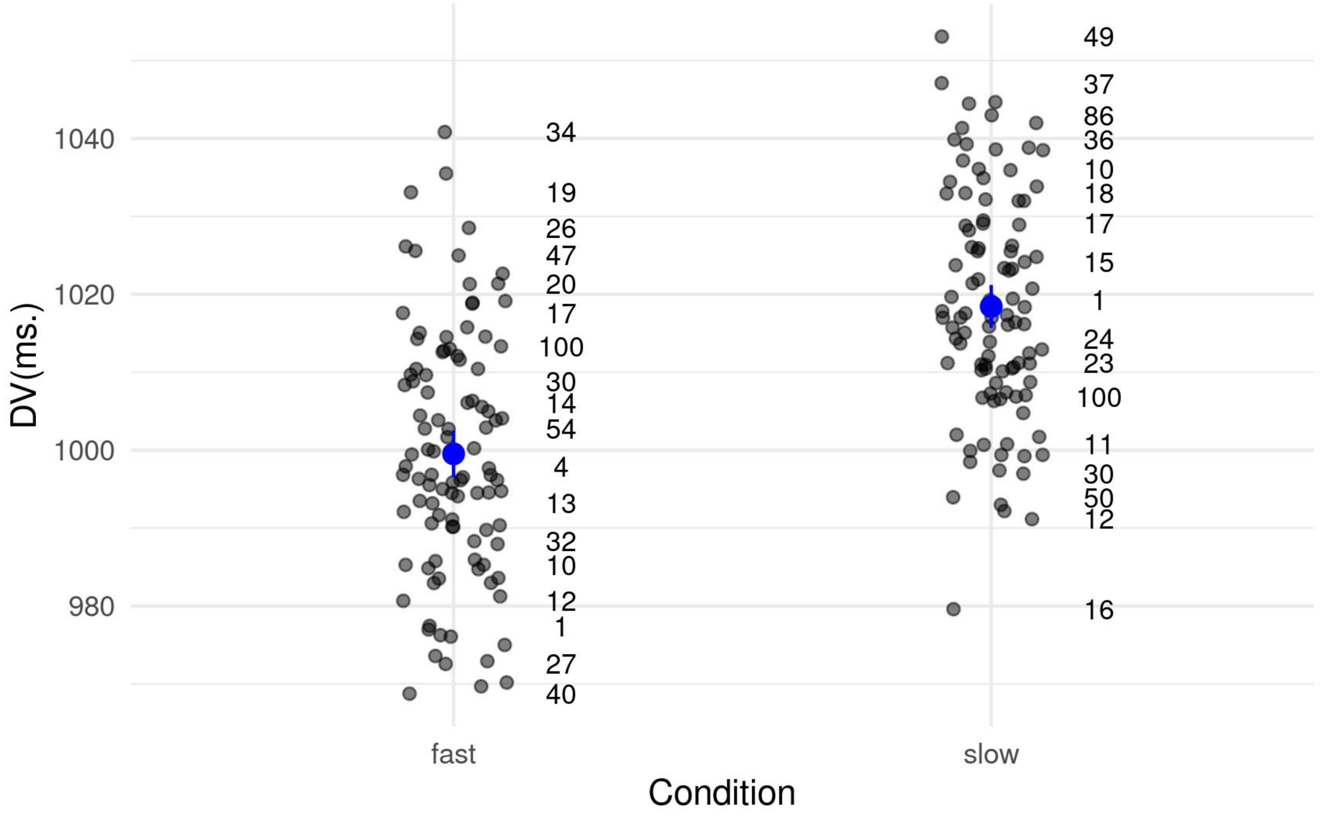


The mean for our “fast” condition is quite close to the population mean (1000), while the mean for our “slow” condition is almost exactly the population mean (1020). Sample 2 has an extreme mean for the ‘slow’ condition - indeed, numerically the ‘slow’ condition is faster than the ‘fast’ condition in Sample 2. This is sampling error in practice and further highlights the problem with small sample sizes...

Simulating 100 experiments

Now imagine a case where we’re simulating data from 100 experiments - each with one between factor with two levels - “Fast” vs. “Slow”.

After we have created our 100 simulations, we can carry out 100 t-tests to determine how many of the simulations produce a significant difference between our two conditions.



Overall the “Slow” condition RTs are higher than the “Fast” Condition RTs - but we can spot some simulations where the difference is negligible or even goes the other way (e.g., Simulation 100). The blue circles corresponds to the overall means and are pretty much bang on 1000 and 1020.

```
result <- NULL
for (i in 1:total_samples) {
  result <- rbind(tidy(t.test(filter(all_tidied_data, condition == "fast" & sample == i)$dv,
                              filter(all_tidied_data, condition == "slow" & sample == i)$dv,
                              paired = FALSE)), result)
}

result

# A tibble: 100 x 10
   estimate estimate1 estimate2 statistic    p.value parameter conf.low conf.high method
     <dbl>     <dbl>     <dbl>     <dbl>    <dbl>        <dbl>     <dbl>     <dbl> <chr>
 1   6.5       1013.     1007.     0.364    0.720        20.2 -30.7      43.7  Welch…
 2   -15       1000.     1015.     -0.695   0.494        22.0 -59.7      29.7  Welch…
 3   7.92      1019.     1011.     0.445    0.661        21.0 -29.1      44.9  Welch…
 4   -16.4     984.      999.      -0.697   0.493        22.0 -65.3      32.5  Welch…
 5   -10.8     1002.     1012.     -0.517   0.612        16.9 -54.7      33.2  Welch…
 6   7.25      1000.     993       0.359    0.723        20.4 -34.8      49.3  Welch…
 7   -35       994.      1030.     -1.66    0.113        20.6 -79.0      8.99 Welch…
 8   -27.5     983       1010.     -1.40    0.175        21.8 -68.2      13.2  Welch…
 9   4.83      1026.     1021.     0.246    0.808        18.3 -36.3      46.0  Welch…
10   -35.8     996.      1032      -1.58    0.130        18.9 -83.2      11.5  Welch…
# … with 90 more rows, and 1 more variable: alternative <chr>
```

How many of the 100 t-tests have produced a p-value < .05?

We can work out for how many of the 100 tests we have found a significant difference at < .05 - and remember, there is actually a real difference (of 20 ms.) in the two population distributions we sampled from!

```
result %>%
  filter(p.value < .05) %>%
  count()

# A tibble: 1 x 1
    n
  <int>
1   17
```

So, 17 times out of 100 - or less than a fifth of the time - we find a significant difference even though one exists in the populations we sampled from. So with a sample size of 24 (12 per group) power to detect the effect we are looking for is 0.17

So let’s work out Cohen’s d as a measure of our effect size - we can do this precisely because we know what the real effect size is comparing the two populations.

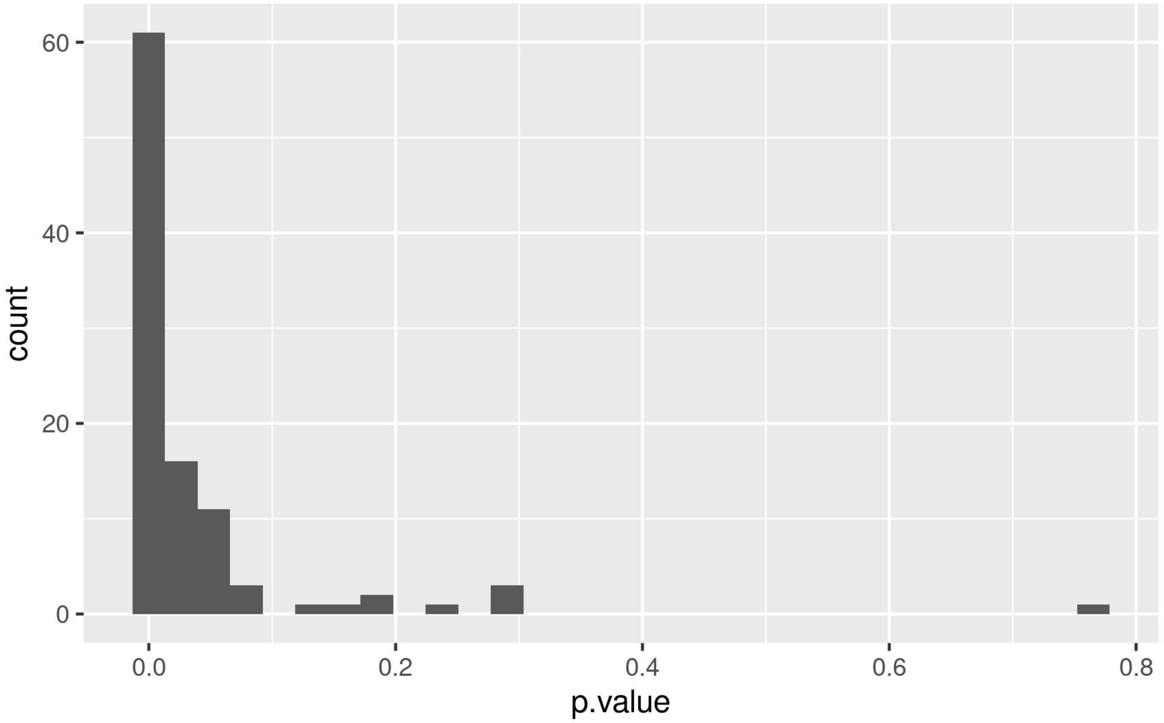
The “classic” Cohen’s d calculation is the mean of one sample minus the mean of the other divided by the pooled standard deviation.

In our case, it’s (1020 - 1000) / 50 which gives a Cohen’s d of 0.4 (which is a small to medium effect size) - standard in many areas of psychology.

We actually need 200(!) participants to give us 80% power for a Cohen’s d = .4

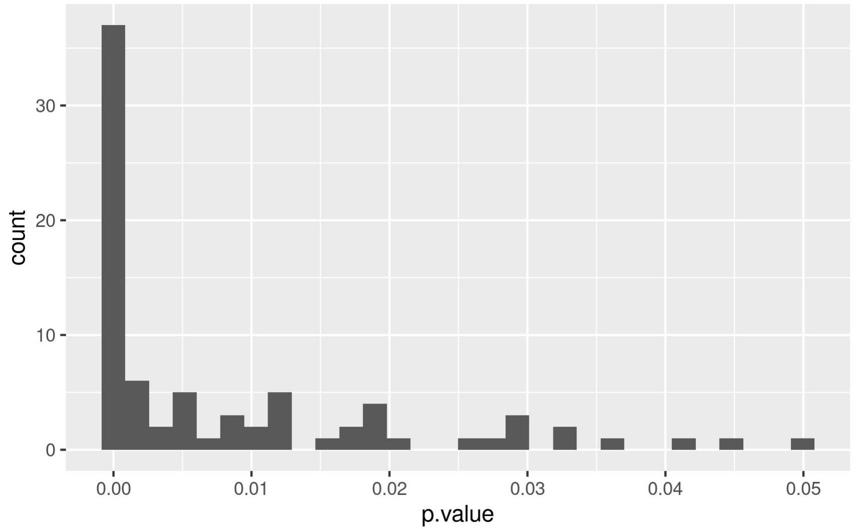
Let’s run the 100 simulations again but this time we’ll set sample size to 200 - here’s the histogram of the p-values - 80 of the t-tests are now significant at < .05:

```
result %>%
  filter(p.value < .05) %>%
  count()

# A tibble: 1 x 1
      n
  <int>
1    80
```

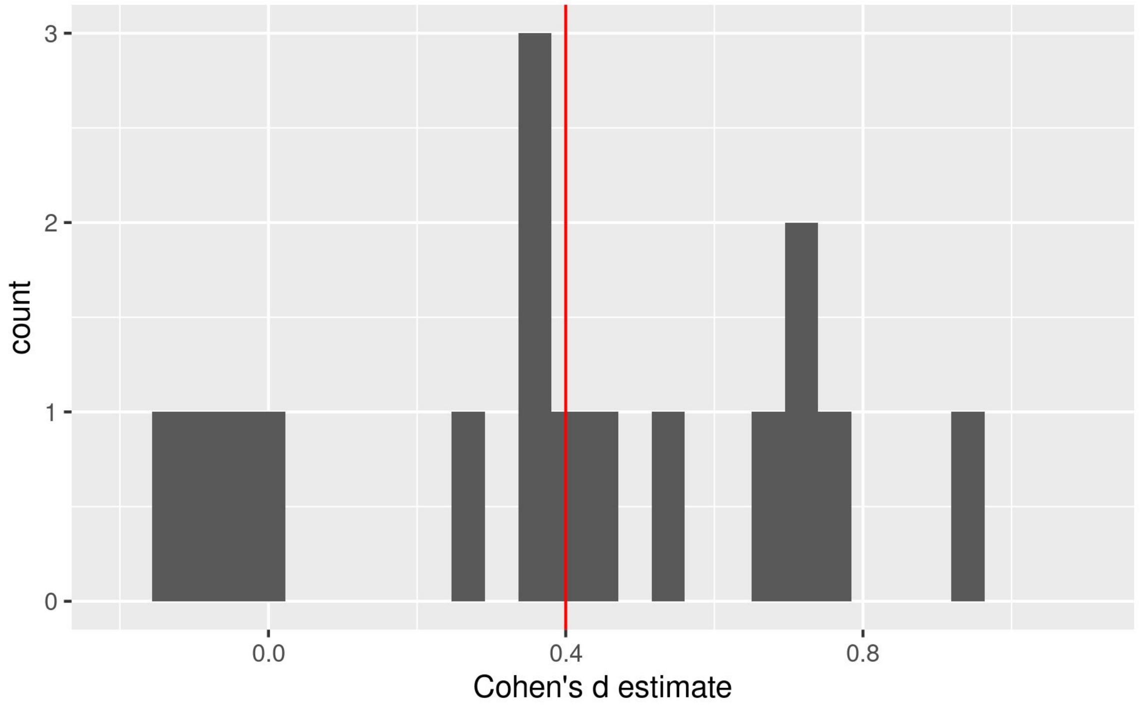


If we zoom in and look at *p*-values < .05 we'll see lots of tiny ones - this is what we would expect to see in the literature but instead analysis has shown more p-values at the threshold of significance (i.e, just below .05) than we would expect - suggests possible *p*-hacking and other QRPs.

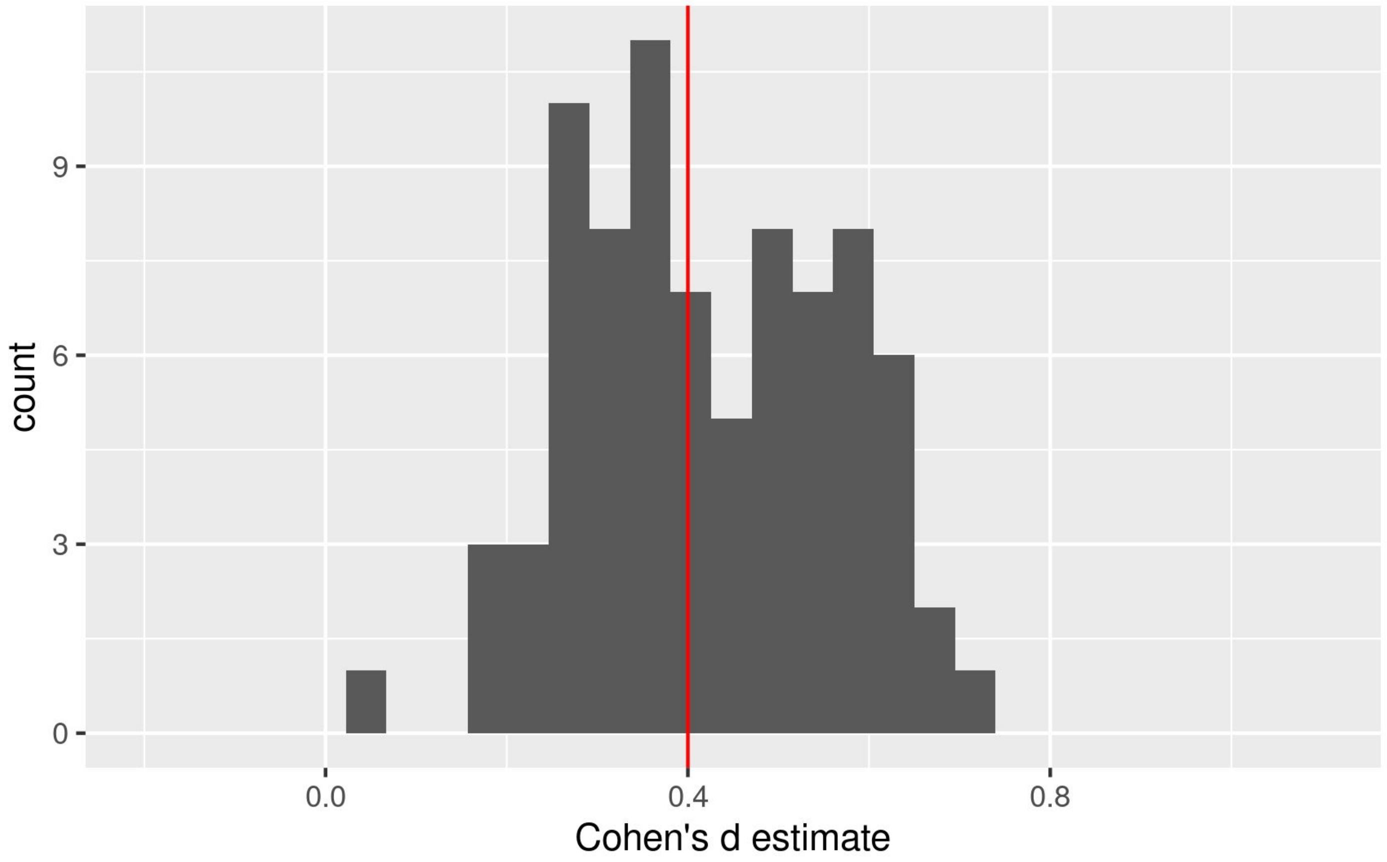
Nathan C. Leggett, Nicole A. Thomas, Tobias Loetscher & Michael E. R. Nicholls (2013). The life of p: "Just significant" results are on the rise. The Quarterly Journal of Experimental Psychology, 66, 2303-2309. http://dx.doi.org/10.1080/17470218.2013.863371



Another problem with small sample studies is not just that they fail to find an effect, but they also provide quite wide estimates of the effect size - here is the histogram of Cohen's d values for the 17 significant results when we have a sample size = 24 with the red vertical line being the true effect size in the population.



And even with a set of well-powered studies (Power = .8) there is still quite a variation in the estimate of the effect size. We need to think of such effect size measures as estimates, rather than points of truth.



Summary

There are many reasons as to why you might want to simulate data before running your study.

You can specify the kind of effect size you're interested in looking for - think about what magnitude might be of theoretical importance - and then model different sample sizes to determine (roughly) how many observations you might need to have a reasonable chance of detecting the effect (assuming it is there).

It's important that the statistical model you're using to test the simulated data is the same type of model you'll be using to analyze the real data.

Data simulation can be used to motivate pre-registered analysis plans (e.g., on OSF).