# Workshop 5 - Data Simulation and Advanced Data Visualisation

Andrew Stewart

Andrew.Stewart@manchester.ac.uk

@ajstewart_lang

https://github.com/ajstewartlang

| Workshop | Topic |
|---|---|
| 1 | Reproducibility and R |
| 2 | General Linear Model (Regression) |
| 3 | General Linear Model (ANOVA) |
| 4 | Mixed Models |
| 5 | Data Simulation and Advanced Data Visualisation |
| 6 | Reproducible Computational Environments and Presentations |

**Assignment**

Assignment to be completed by Semester 1 exam period.

# Today

- You’re going to learn how to simulate 100s (or even 100,000s) of repeats of the same experiment to determine what pattern of *p*-values and effect size estimates you'll get (assuming there is a medium size effect to be found).

- We’ll look at how to use loops to do this, and how to package a set of code that you might find yourself re-using as a function.

- We’re also going to look at more advanced data visualisation tools including animated plots using the `gganimate` package, visualisations of text data, sentiment analysis using `tidytext`, how to create BBC-style visualisations using `ggplot2` and the BBC R Cookbook, and how to use git for Version Control.

# Writing clear, human readable, and consistent code.

- I’ve seen a variety of different types of approaches to coding in assignments - while most code works fine, it can vary quite a lot in terms of clarity and consistency.

- Code needs to be both computer readable and human readable.

- A good resource for mastering writing clear and human readable code can be found in the Tidyverse style guide:

  `https://style.tidyverse.org/_main.pdf`

# Simulating Data in R

- So far we've looked at wrangling, visualising, and modelling data using R.
- Now we're going to look at creating or simulating data using R and looking at some programming in R.

# 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 on OSF).

    - 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  1.12242126  2.13538261 -0.04670306
0.32518029

> 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…

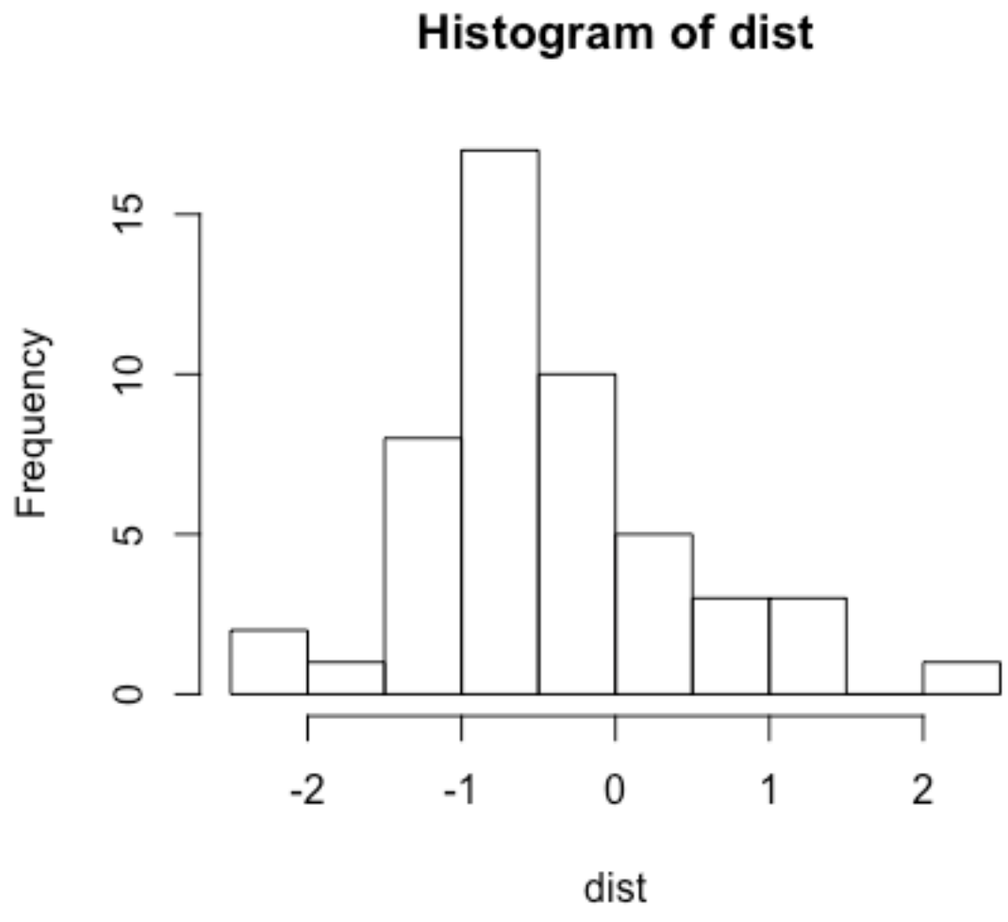
- We want to make sure we can replicate our sample - we can use the `set.seed()` function to specify the seed of the randomisation (so we can rerun the code and get the same result).

```
> 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.

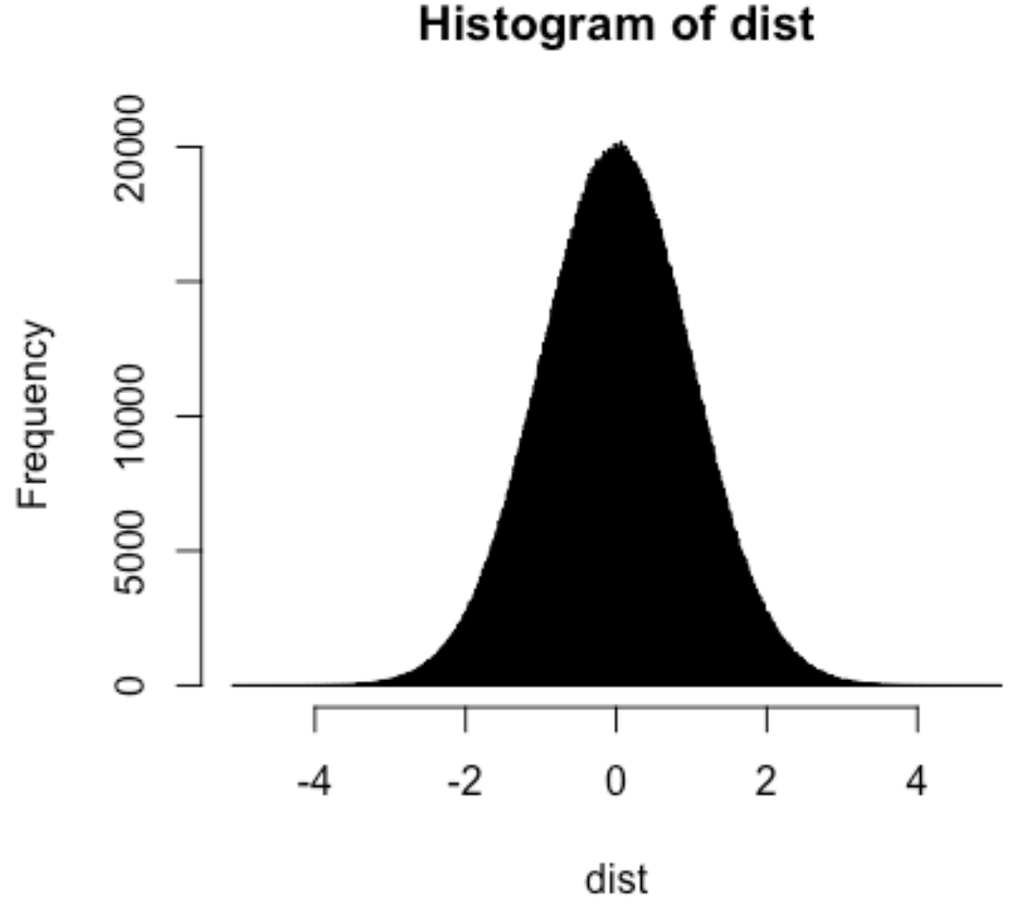
- We can map our random sample onto a new variable I”m calling `dist` and then plot a histogram of the values. Here is a N = 50 sample.

```
> set.seed(1234)

> dist <- rnorm(50, 0, 1)

> hist(dist)
```

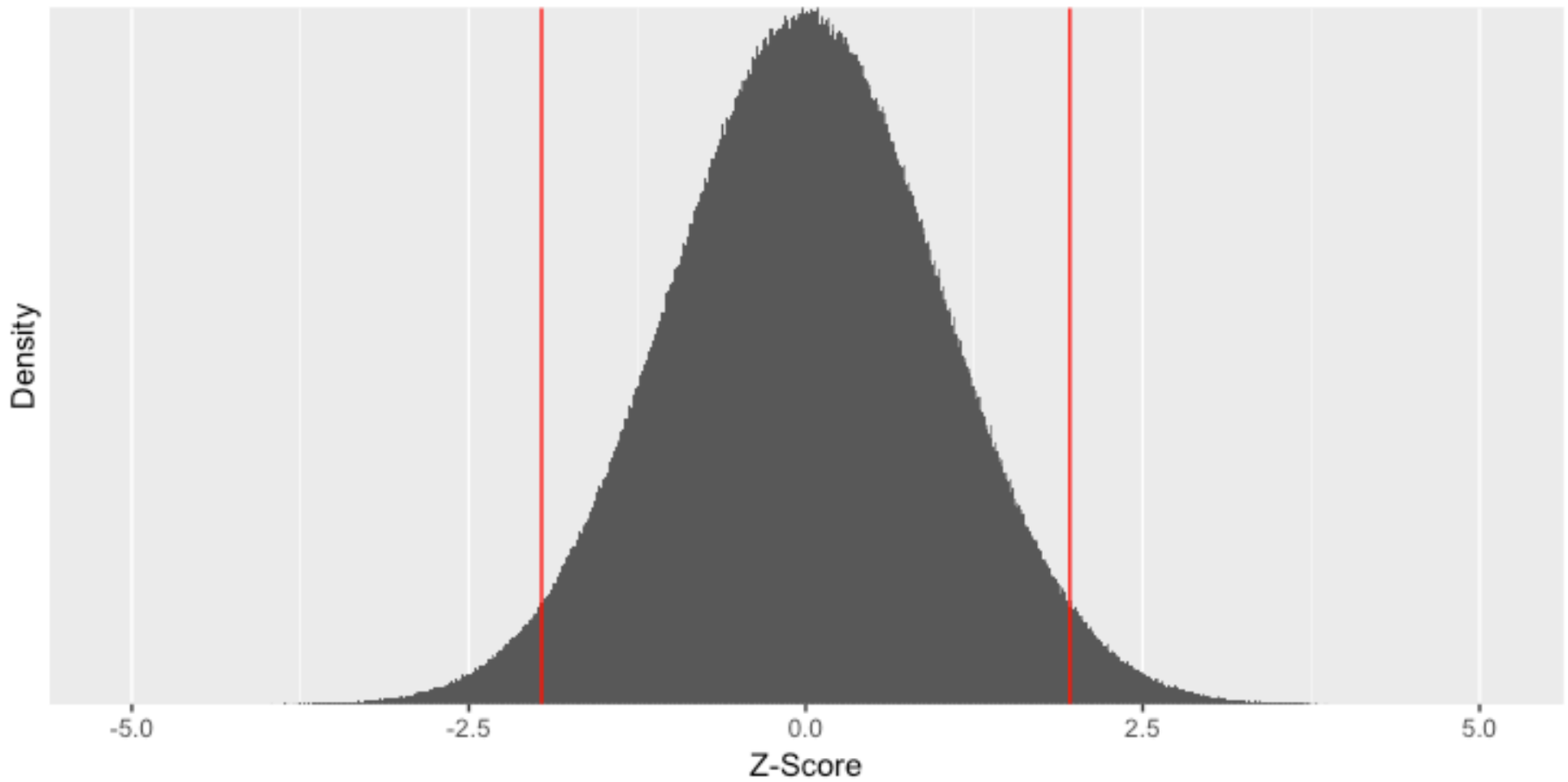


- The larger the sample the more representative it is of the population it is drawn from. Here is a N = 5 million sample from the standard normal distribution.

```
> set.seed(1234)

> dist <- rnorm(5000000, 0, 1)

> hist(dist, breaks = 1000)
```



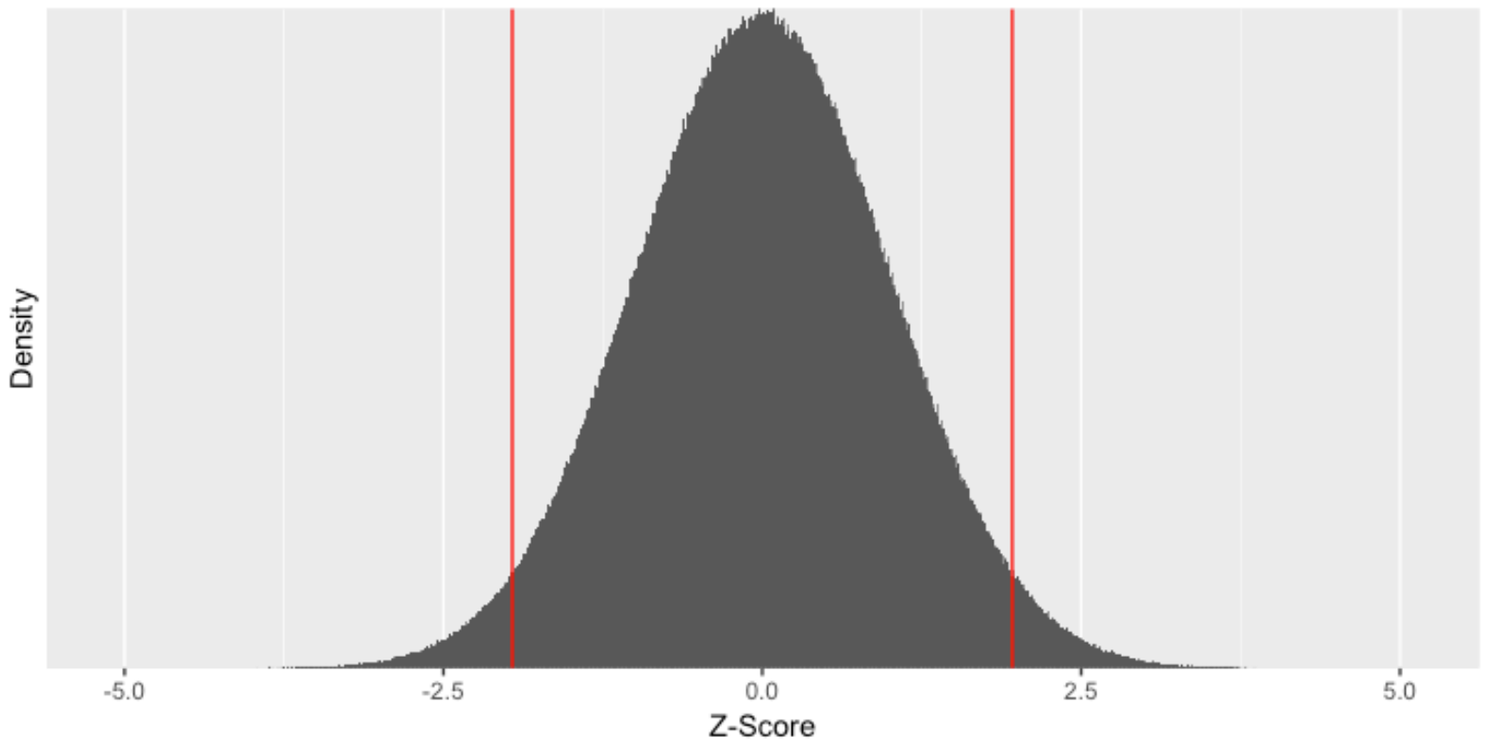
- In the standard normal distribution, 95% of the data is contained with 1.96 standard deviations either side of the mean.



- We can use the `pnorm()` function to give us the area under the curve of the normal distribution.

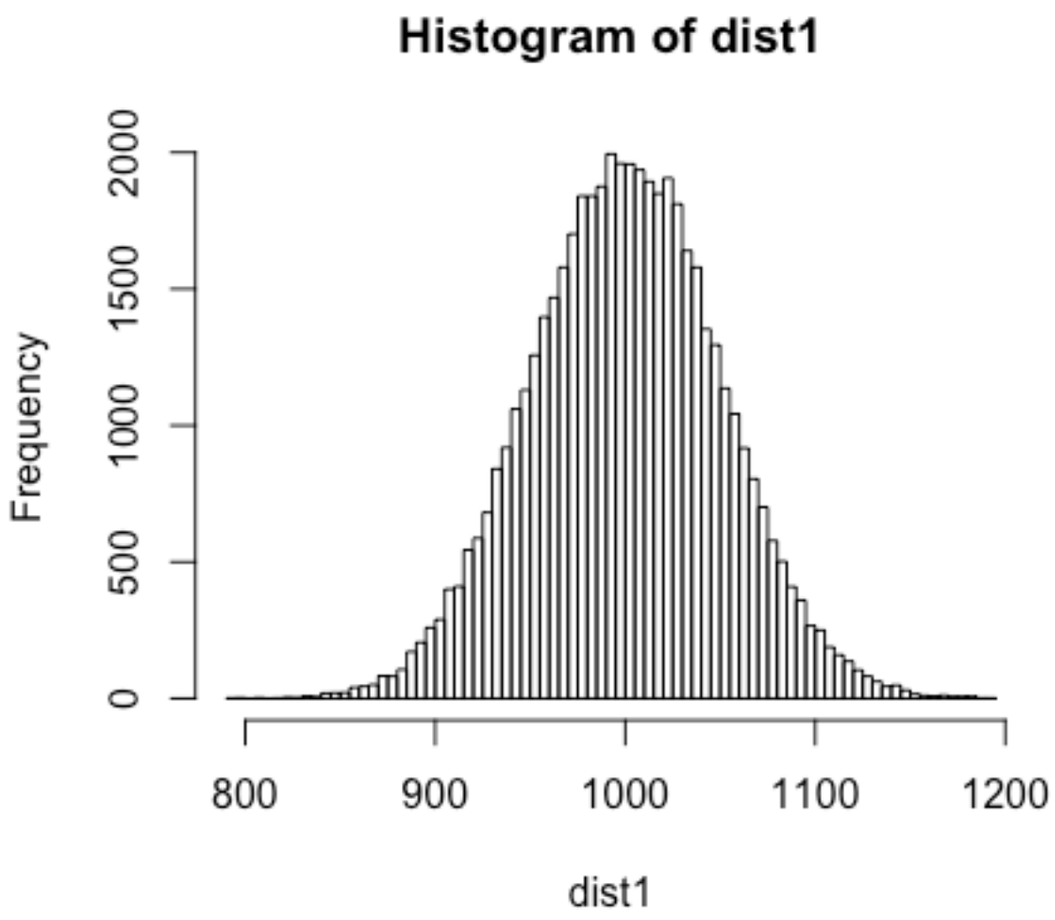
- To work out the area bounded by 1.96 standard deviations from the mean in the standard normal distribution:

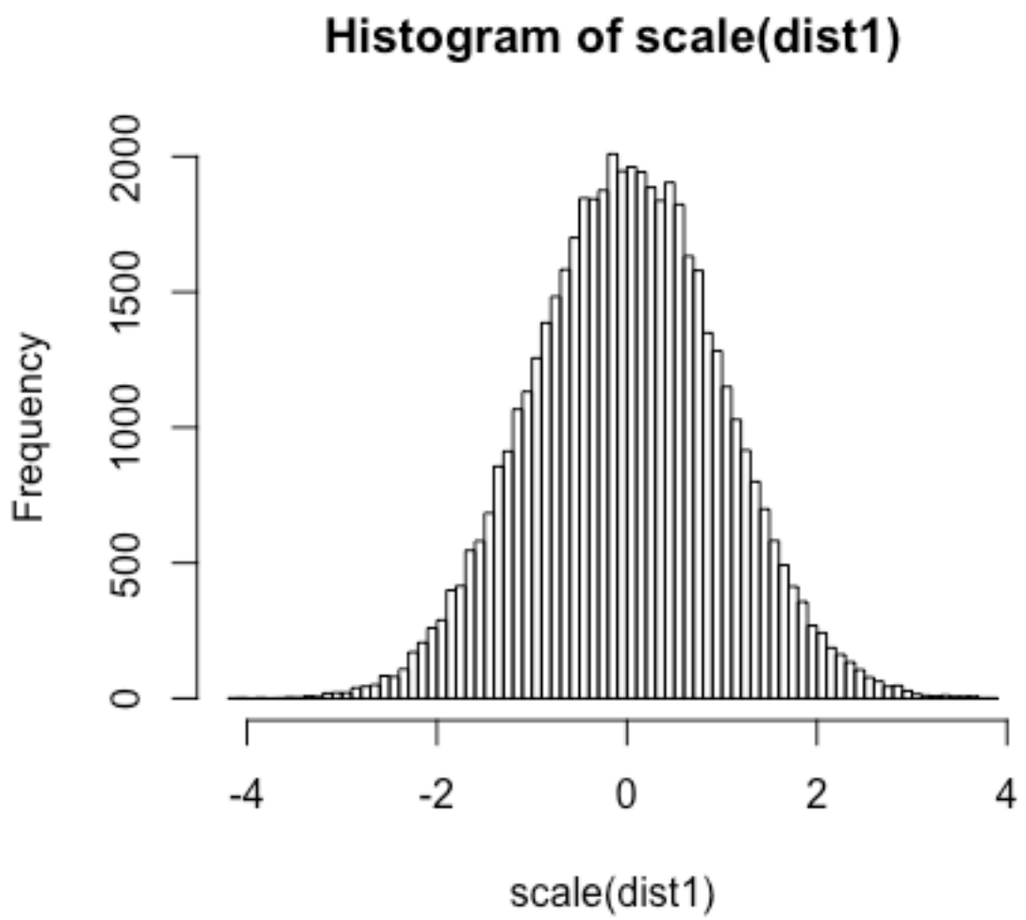
```
> pnorm(1.96, mean = 0, sd = 1) - pnorm(-1.96, mean = 0, sd = 1)

[1] 0.9500042
```



- We can convert any normally distributed data to the standard normal (i.e., Z) distribution by using the `scale()` function. This will centre it so the mean is 0 and scale it so that the standard deviation is 1.

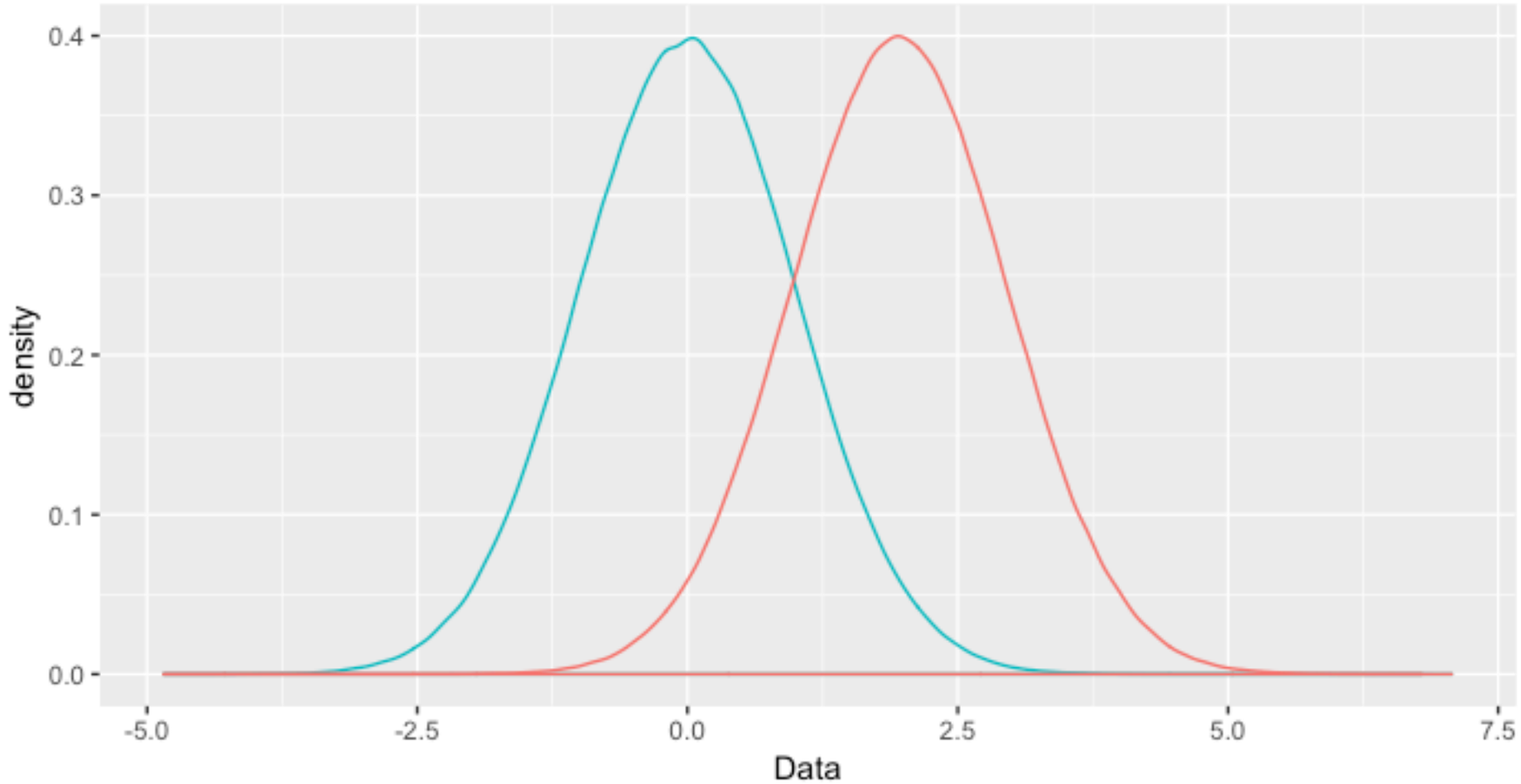
```
> set.seed(1234)

> dist1 <- rnorm(50000, 1000, 50)

> hist(dist1, breaks = 100)

> hist(scale(dist1), breaks = 100)
```



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

```
> set.seed(1234)
> cond1 <- rnorm(1000000, 0, 1)
> cond2 <- rnorm(1000000, 1.96, 1)
> data <- as.tibble(cbind(cond1, cond2))

> ggplot(data) +
    geom_density(aes(x = cond1, y = ..density.., colour = "red")) +
    geom_density(aes(x = cond2, y = ..density.., colour = "green")) +
    xlab("Data") +
    guides(colour = FALSE)
```



# Some useful functions

- 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
```

# Some useful functions

- 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
```

# Some useful functions

- 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
```

- `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
```

- 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" "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" "fast"

[13] "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow" "slow"
"slow" "slow" "slow"
```

- 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
```

- 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"
```

- 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’).

- 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
```

- 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 I'm calling `data`.

- A tibble is really just a supercharged dataframe.

```
> data <- as.tibble(cbind(participant, condition, dv))
> 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
```

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

```
> data$condition <- as.factor(data$condition)
> data$dv <- as.integer(data$dv)
> 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
```

- 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(data, aes(x = condition, y = dv, fill = condition)) +
geom_violin() +
stat_summary(fun.data = "mean_cl_boot", colour = "black") +
geom_jitter(alpha = .2, width = .05) +
guides(fill = FALSE)
```


```
data %>%
  group_by(condition) %>%
  summarise(Mean = mean(dv), SD = sd(dv))

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

- Looks pretty much like what we'd expect to me given a bit sampling error…

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

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

 Welch Two Sample t-test

data:  filter(data, condition == "fast")$dv and filter(data, condition == "slow")$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?

- Note the default t-test in R is Welch's t-test (rather than Student's) - do you know why?

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

```
> result <- tidy(t.test(filter(data, condition == “fast")$dv, filter(data, condition == "slow")$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?

- We can, but first we need to learn about loops…

# Loops

- Loops involve repeating a command or a set of commands surrounded by `{}` a certain number of times - popular with kids in the 1980s…

- Printing something `i` times:

```
> for (i in 1:10){
>   print("Hello world")
> }
```

```
[1] "Hello world"
[1] "Hello world"
[1] "Hello world"
[1] "Hello world"
[1] "Hello world"
[1] "Hello world"
[1] "Hello world"
[1] "Hello world"
[1] "Hello world"
[1] "Hello world"
[1] "Hello world”
```

- Our variable `i` increments one by one from the start of the sequence `1:10` to the end.

- We can also use `i` in our code embedded within the loop:

```
> 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 also add `i` as part of a string:

```
> for (i in 1:10){
>   print(paste("This is the number", i, sep=" "))
> }
[1] "This is the number 1"
[1] "This is the number 2"
[1] "This is the number 3"
[1] "This is the number 4"
[1] "This is the number 5"
[1] "This is the number 6"
[1] "This is the number 7"
[1] "This is the number 8"
[1] "This is the number 9"
[1] "This is the number 10”
```

- We can use `i` to index a column in a tibble - here is the tibble `data`:

```
 > 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
```

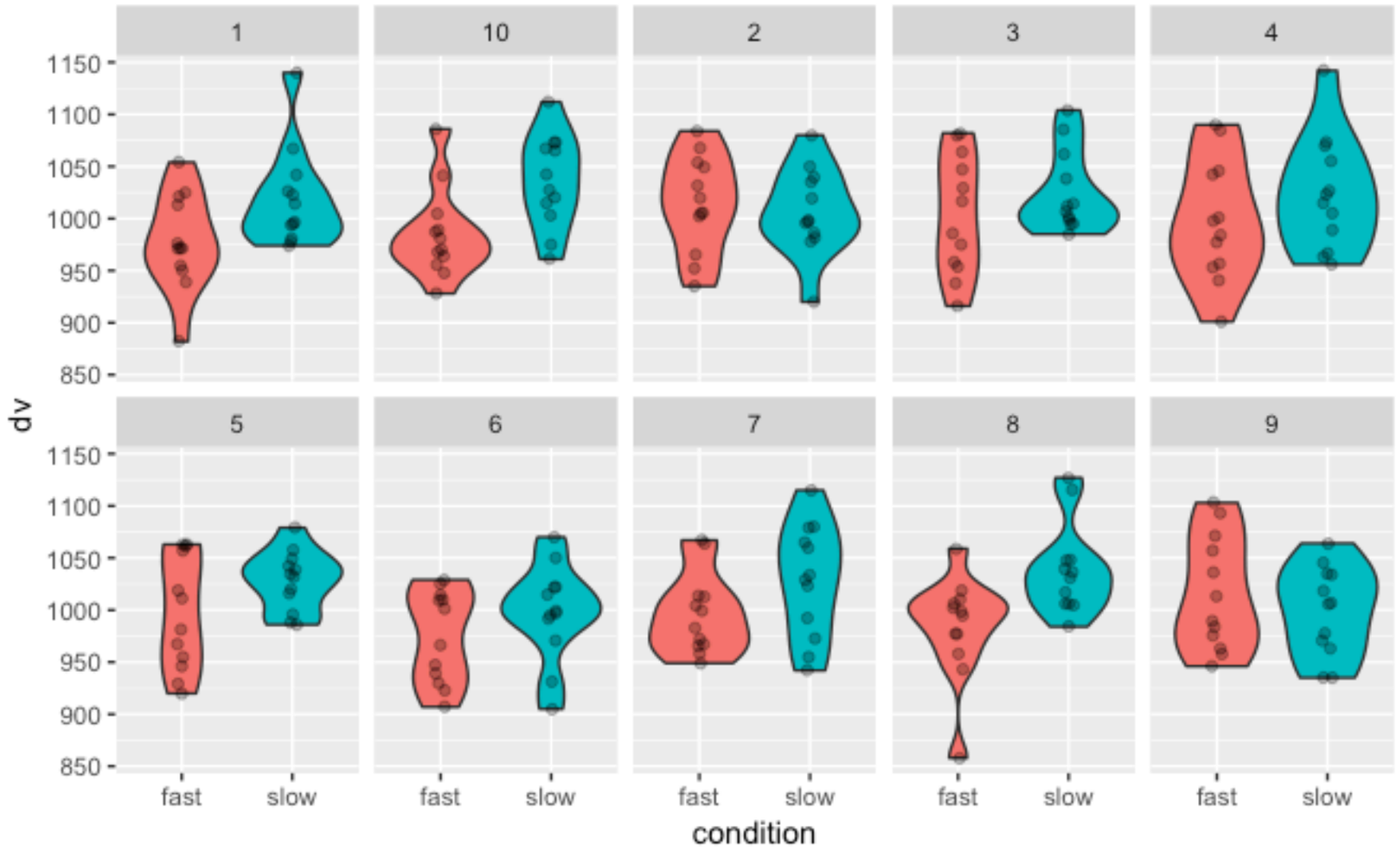
- And here we use `i` to index the column dv in the tibble called `data`:

```
> for (i in 1:10){
>  print(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:

```
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))
  data <- as.tibble(cbind(participant, condition, dv, sample))
  all_data <- rbind(data, all_data)
}

all_data$condition <- as.factor(all_data$condition)

all_data$dv <- as.integer(all_data$dv)
```

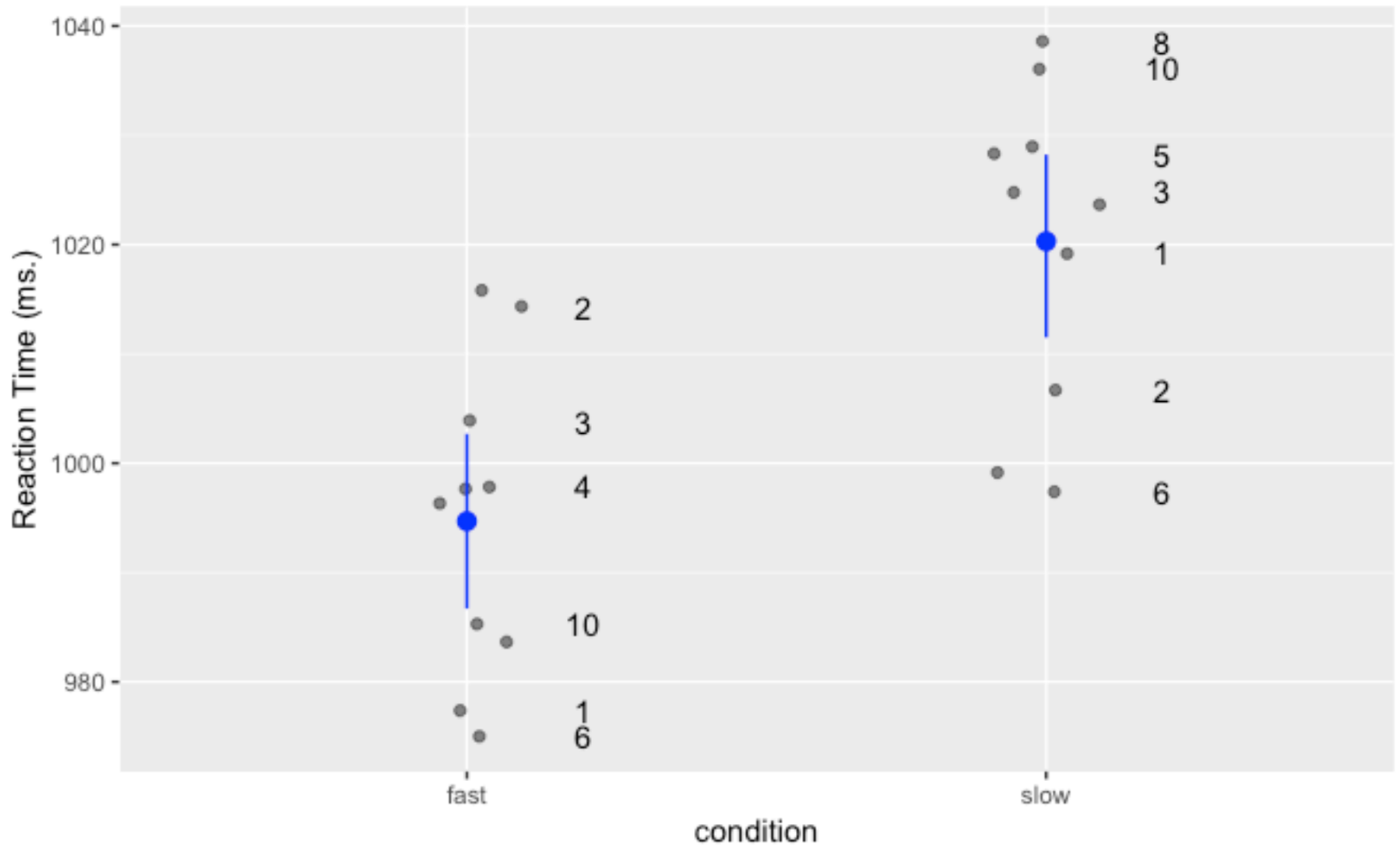
```
ggplot(all_data, aes(x = condition, y = dv, fill = condition)) +
  geom_violin() + geom_jitter(alpha = .3, width = .05) +
  guides(fill = FALSE) + facet_wrap(~sample, ncol = 5, nrow = 2)
```

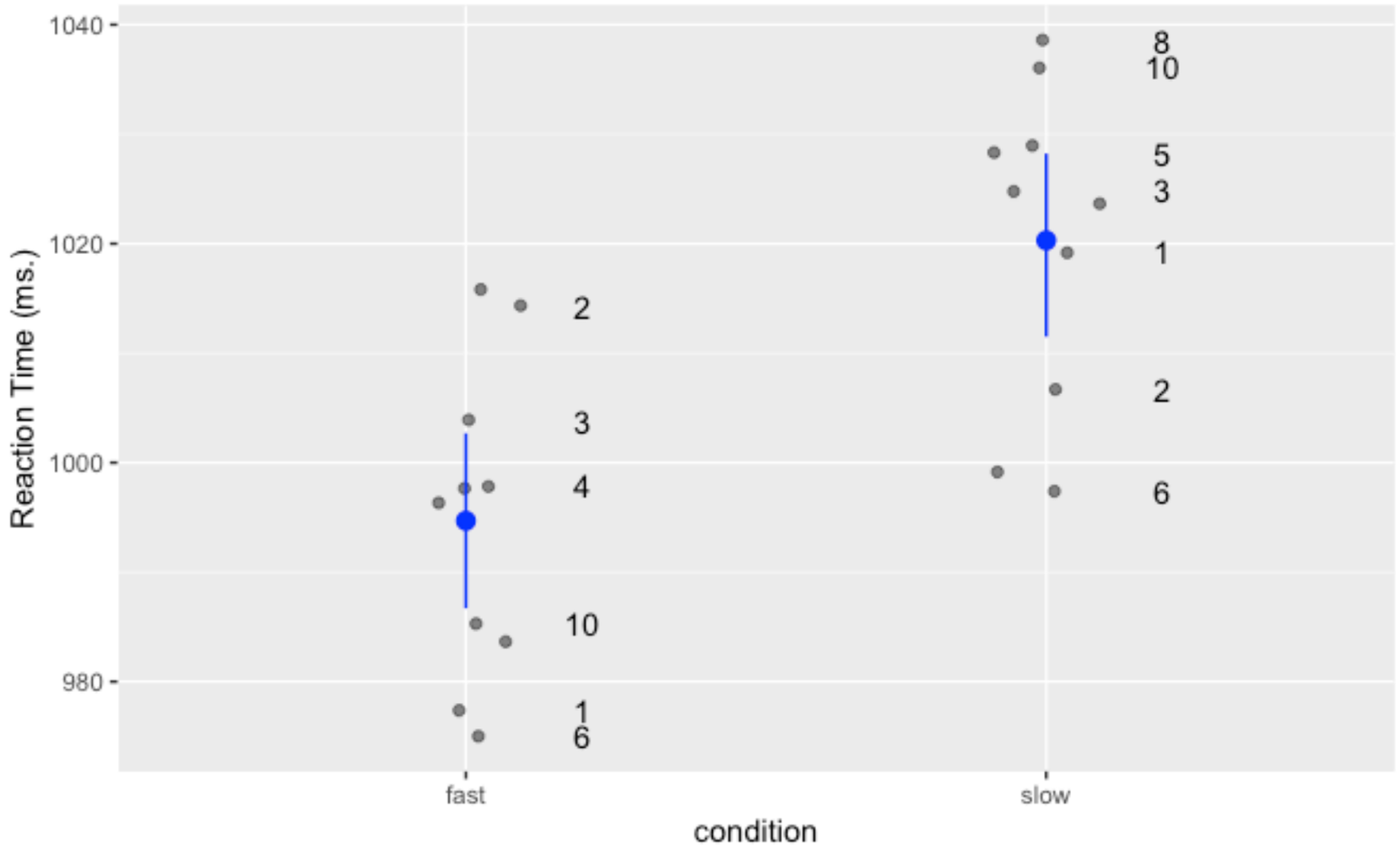



```
> str(all_data)
Classes ‘tbl_df’, ‘tbl’ and 'data.frame': 240 obs. of  4 variables:
 $ participant: chr  "1" "2" "3" "4" ...
 $ condition  : Factor w/ 2 levels "fast","slow": 1 1 1 1 1 1 1 1 1 1 ...
 $ dv         : int  981 948 964 1041 989 970 956 968 1086 928 ...
 $ sample     : chr  "10" "10" "10" "10" ...
```

```
> all_data %>% group_by(condition, sample) %>% summarise(mean(dv), sd(dv))

# A tibble: 20 x 4
# Groups:   condition [?]
   condition sample `mean(dv)` `sd(dv)`
   <fct>     <chr>       <dbl>    <dbl>
 1 fast      1            977.     46.0
 2 fast      10           985.     42.8
 3 fast      2           1014.     46.5
 4 fast      3           1004.     57.2
 5 fast      4            998.     58.2
 6 fast      5            998.     54.9
 7 fast      6            975      44.2
 8 fast      7            996.     38.7
 9 fast      8            984.     49.6
10 fast      9           1016.     54.8
11 slow      1           1019.     47.1
12 slow      10          1036.     44.3
13 slow      2           1007.     41.5
14 slow      3           1025.     38.8
15 slow      4           1024.     54.5
16 slow      5           1028.     28.5
17 slow      6            997.     46.0
18 slow      7           1029.     54.2
19 slow      8           1038.     43.3
20 slow      9            999.     42.7
```

```
all_data %>%
  group_by(condition, sample) %>%
  summarise(average = mean(dv), sd(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")
```



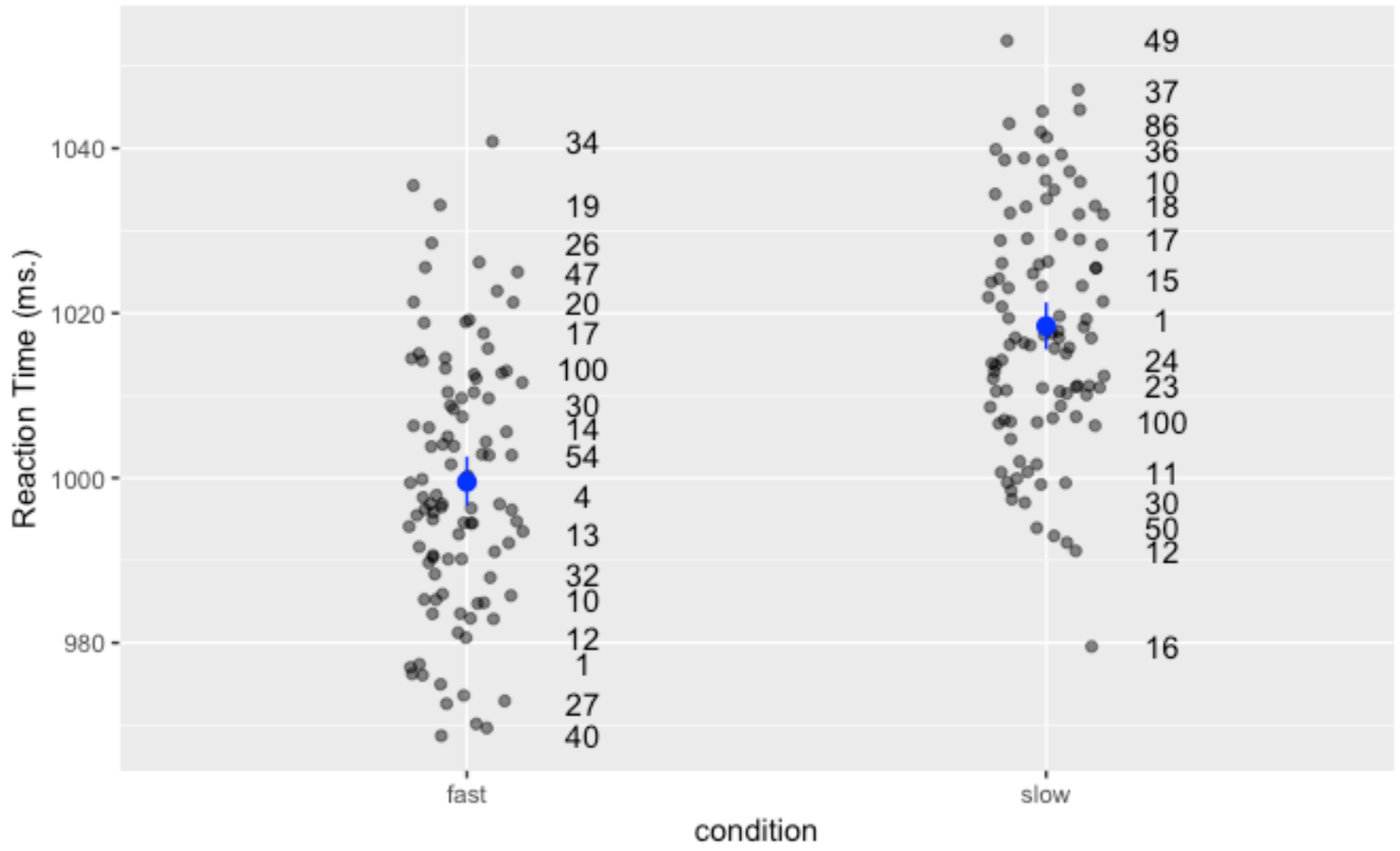



The mean for our ‘fast’ condition is a bit away close to the population mean (1000), while the mean for our ‘slow’ condition is very close to the population mean (1020). Sample 2 has an extreme mean for the ‘slow’ condition which is having an adverse effect on the overall mean for this condition - indeed, 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…

# Comparing conditions

- Now imagine a case where we're simulating data from 100 experiments - each with one repeated measures 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 wrong way (e.g., Simulation 100). The blue circles corresponds to the overall means and are pretty much bang on 1000 and 1020.

- We can use another loop to run `i` number of independent sample t-tests and to save the results of each test to a new data frame we are calling `result`

```
result <- NULL

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

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

- 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!

```
> count(filter(result, p.value < .05))
# A tibble: 1 x 1
      n
  <int>
1    17
```

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

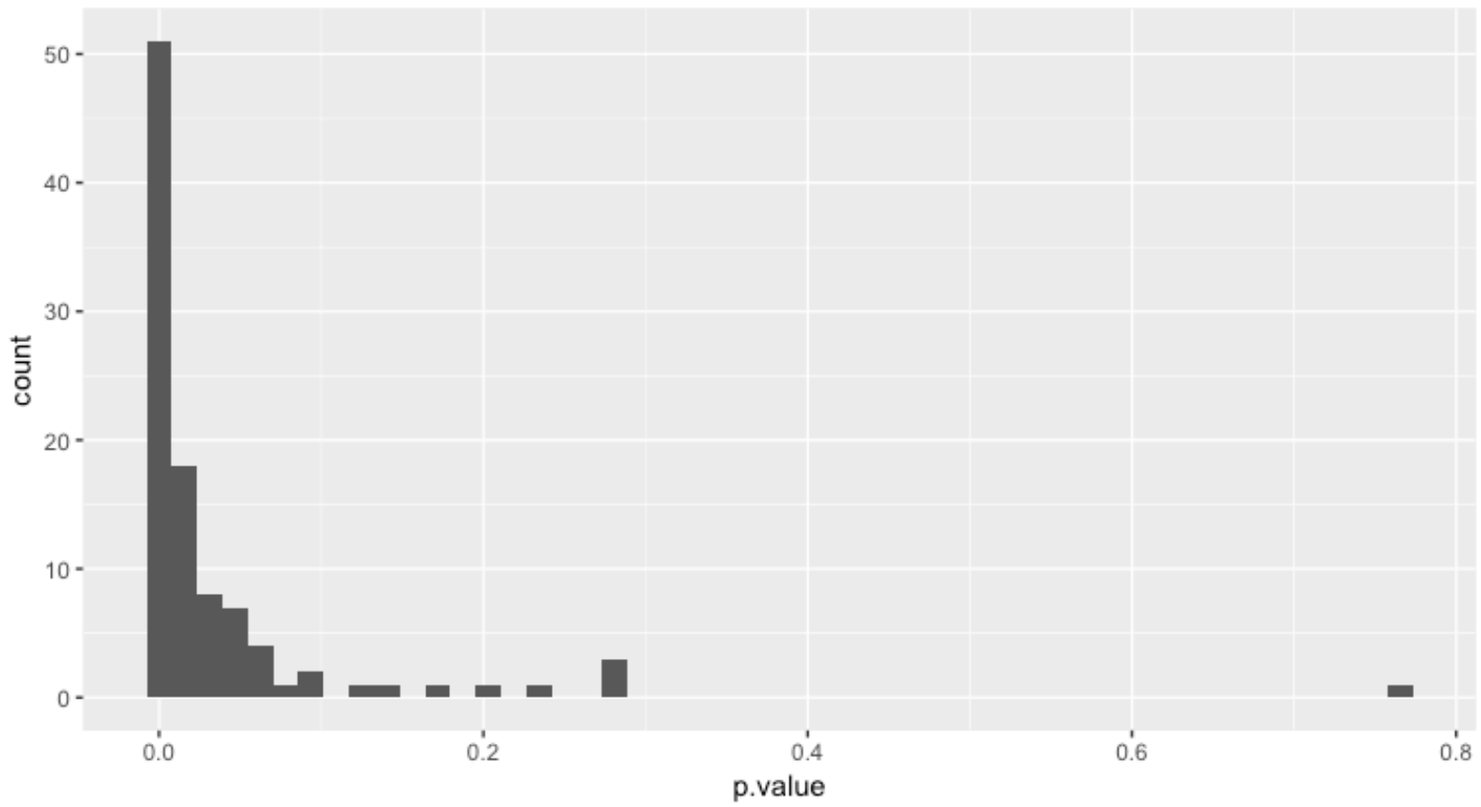
- So let’s work out Cohen’s d as a measure of our effect size - we can do this precisely because we know what the real effect size is comparing the two populations.

- 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

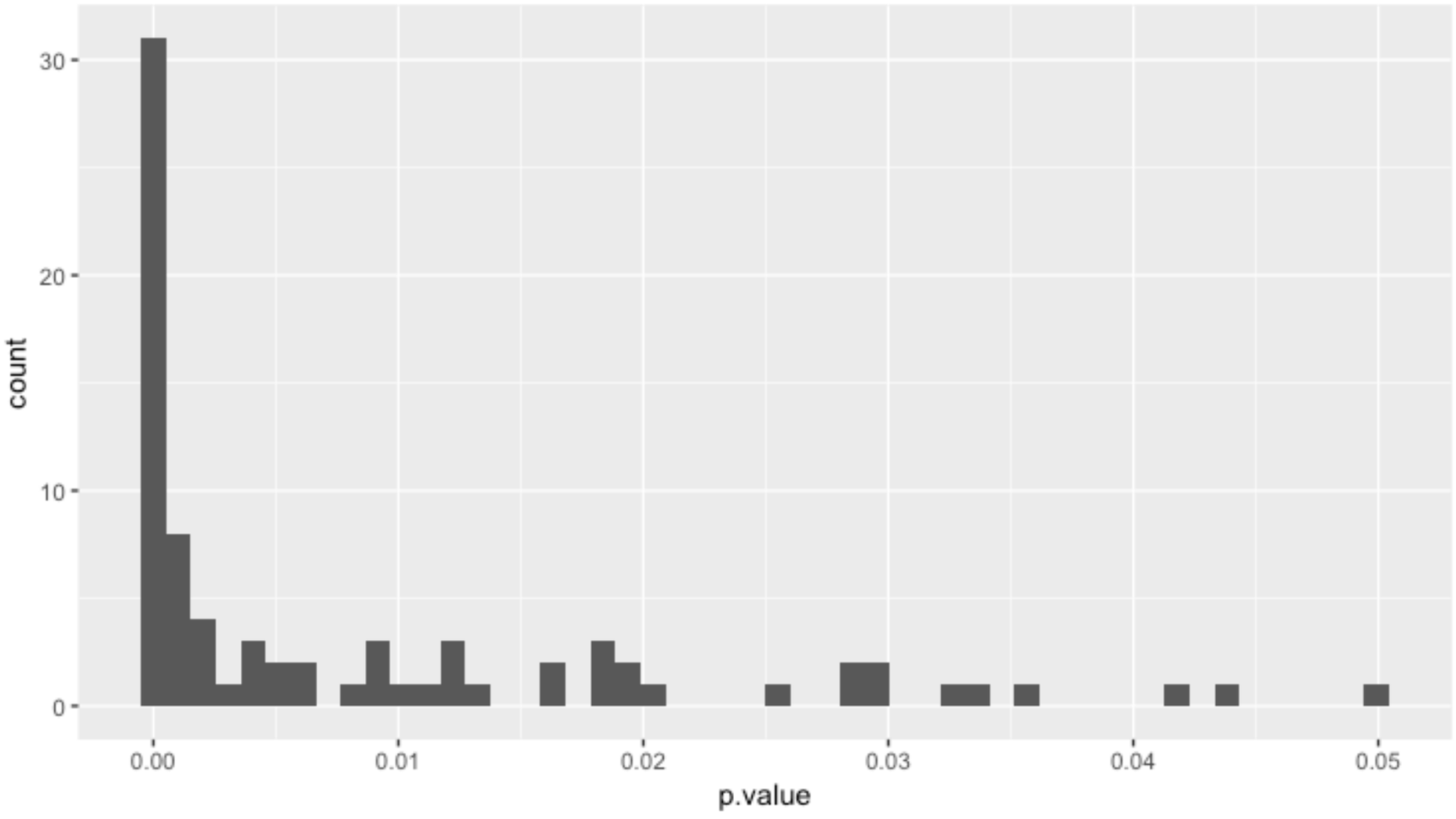
- Let's run the 100 simulations again but this time we'll set sample size to 200 - here's the histogram of the p-values - 80 of the t-tests are now significant at < .05:



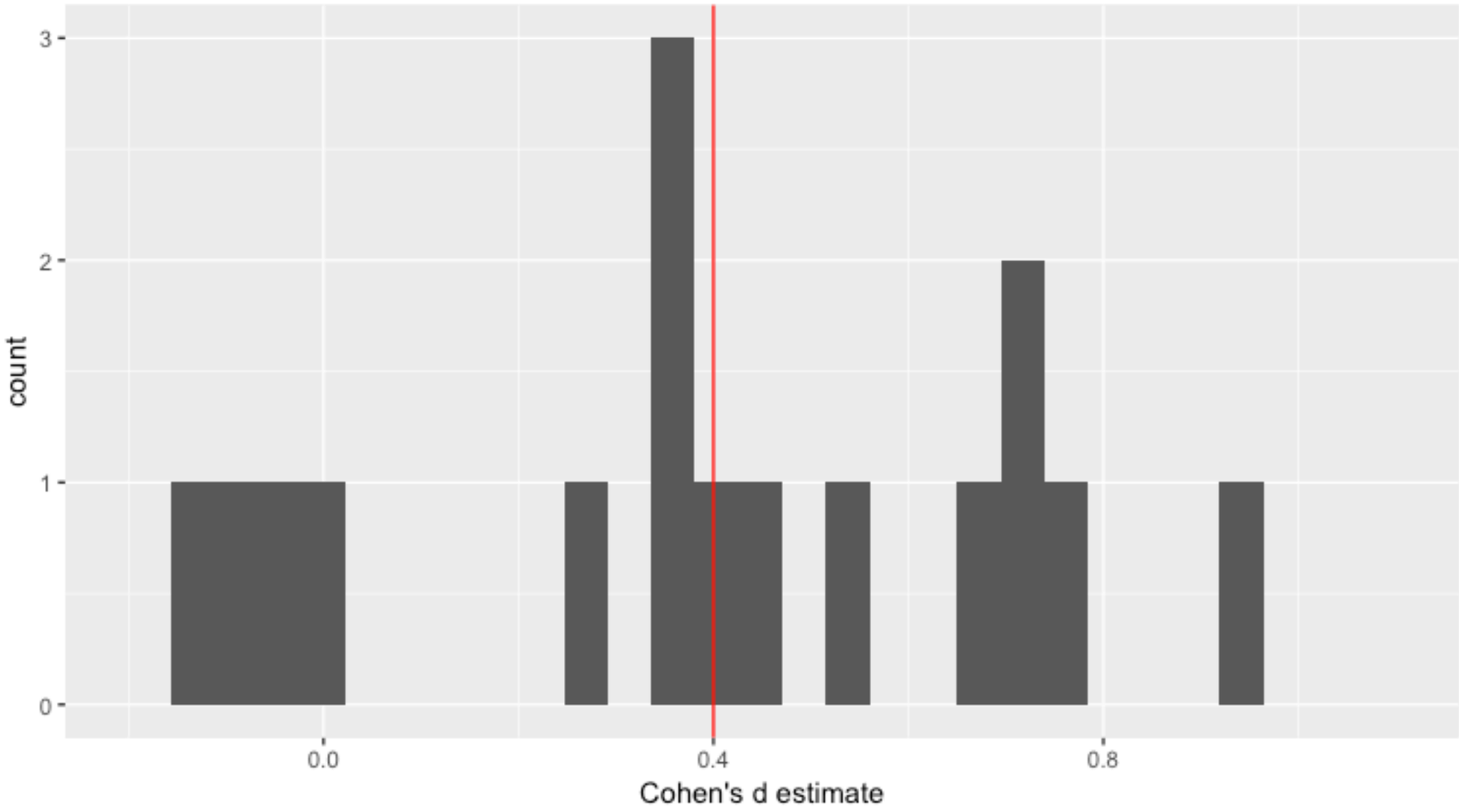

```
> count(filter(result, p.value < .05))
# A tibble: 1 x 1
      n
  <int>
1    80
```

- 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.

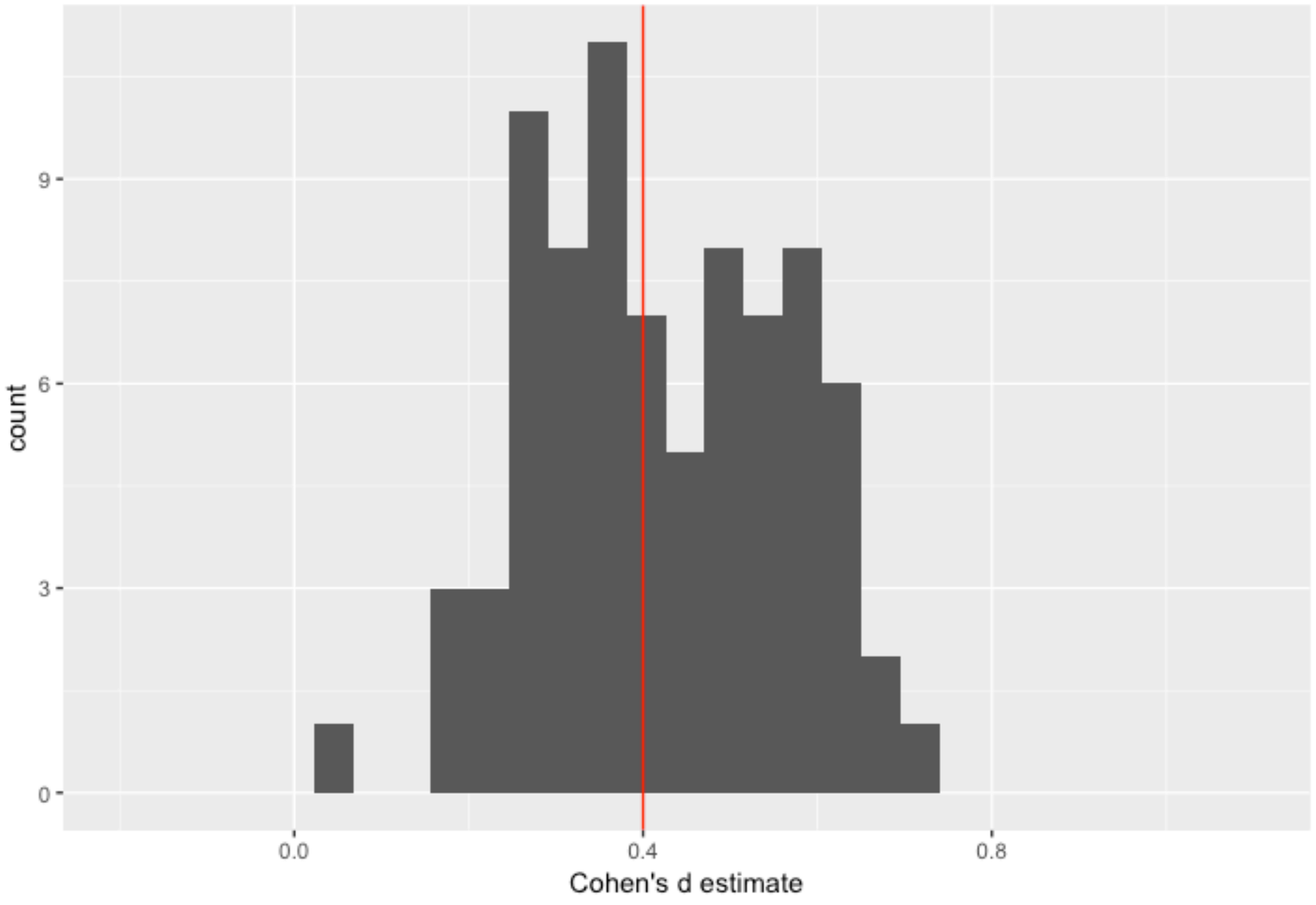
- Nathan C. Leggett, Nicole A. Thomas, Tobias Loetscher & Michael E. R. Nicholls (2013). The life of p: "Just significant" results are on the rise. *The Quarterly Journal of Experimental Psychology, 66,* 2303-2309. http://dx.doi.org/10.1080/17470218.2013.863371



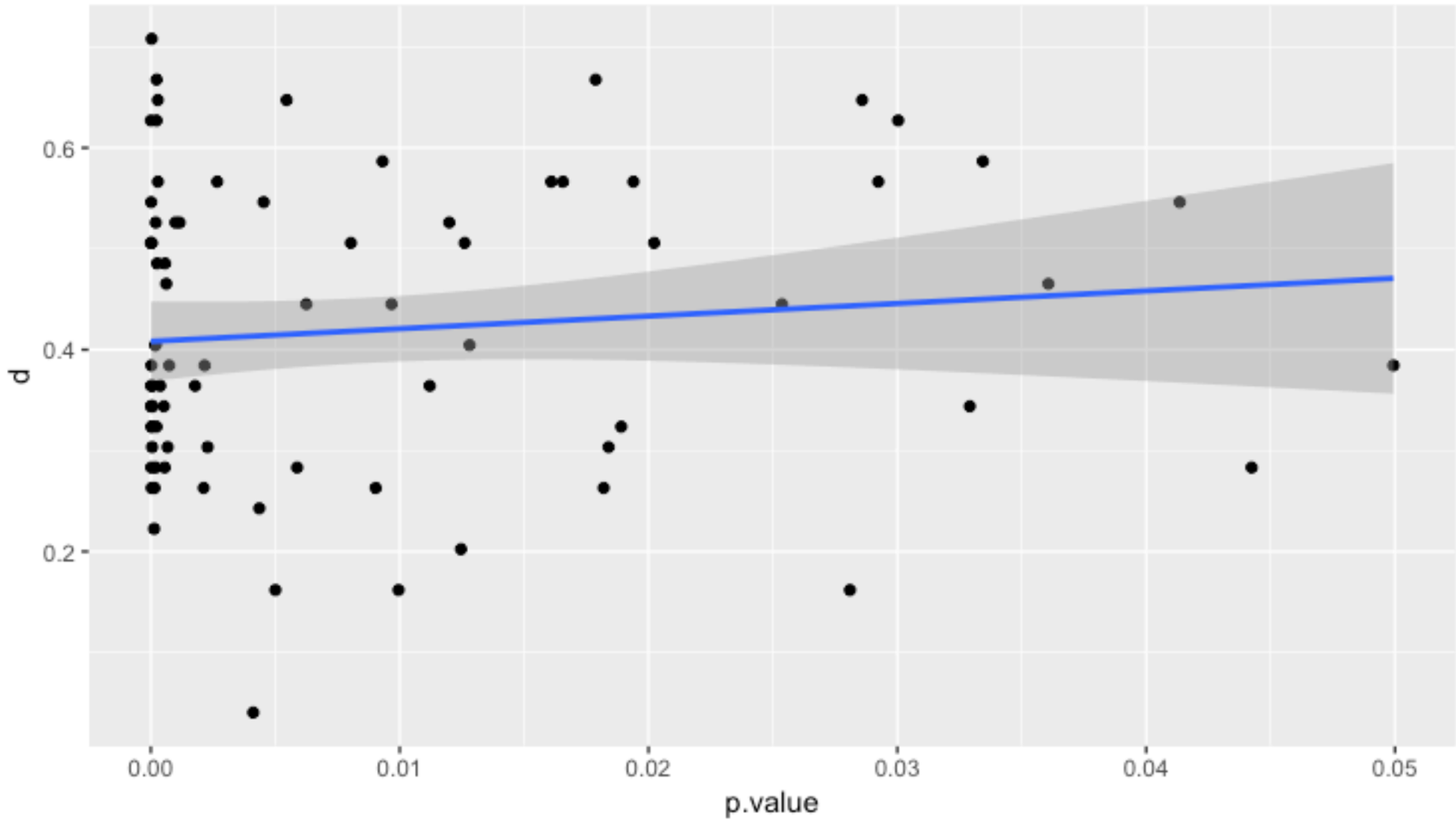
- Another problem with small sample studies is not just that they fail to find an effect, but they also provide quite wide estimates of the effect size - here is the histogram of Cohen's d values for the 17 significant results when we have a sample size = 24 with the red vertical line being the true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



- When we increase the sample size to 200 (for .8 power) we get a much more accurate (and less variable) view of the effect size - but each effect size estimate is still just one point drawn from a distribution of effect sizes .



- It’s worth noting, there’s no clear relationship between the p-value of a t-test and Cohen’s d (i.e., smaller p-values don’t mean bigger effect size estimates). Pearson’s r in this case = 0.11



# The Advantage of Simulation

- 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).

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

- Wouldn't it be handy if we could easily change our sample size parameter without having to dig around in our code?

# Writing Functions in R

- As a general rule of thumb, if you ever want to run the same code more than once (but with different parameter values) you might want to consider turning the code into a function…

- Most of the commands you use in R are actually functions. A function takes a set of inputs, and produces an output.

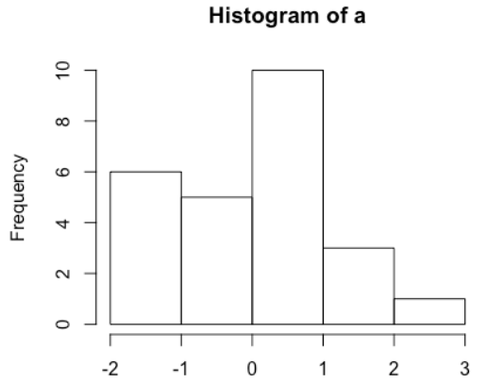
- On the following slide I have taken the code I wrote to sample from the standard normal distribution and have turned it into a function which takes as its input the sample size you want to simulate.

```
just_simulate <- function(sample_size) {
    a <- rnorm(sample_size, 0, 1)
    hist(a)
}
```

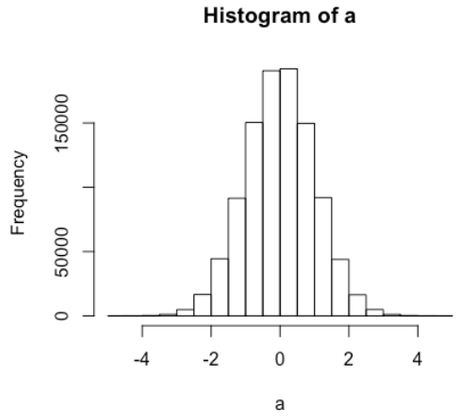
- I have created a function I've called `just_simulate()` which takes one parameter (which I've called `sample_size`), samples this number of times from the standard normal distribution, and then plots a histogram of the sample.

- I ask the function to simulate first 25 and then 1000000 samples simply by typing:

```
> just_simulate(25)
```




```
> just_simulate(1000000)
```



- We can change any set of R code into a function in a similar way.

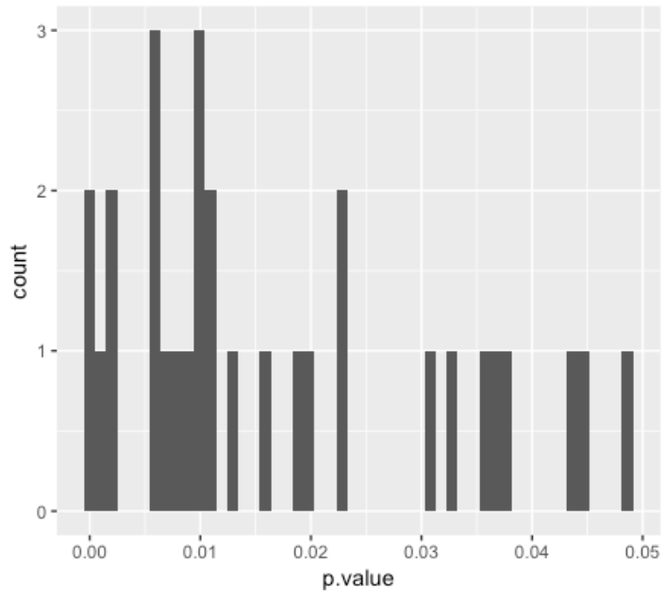
- The code for generating 100 samples, running the t-tests, and then plotting the results of the t-tests is turned into a function on the next slide - I've called it `simulate()`

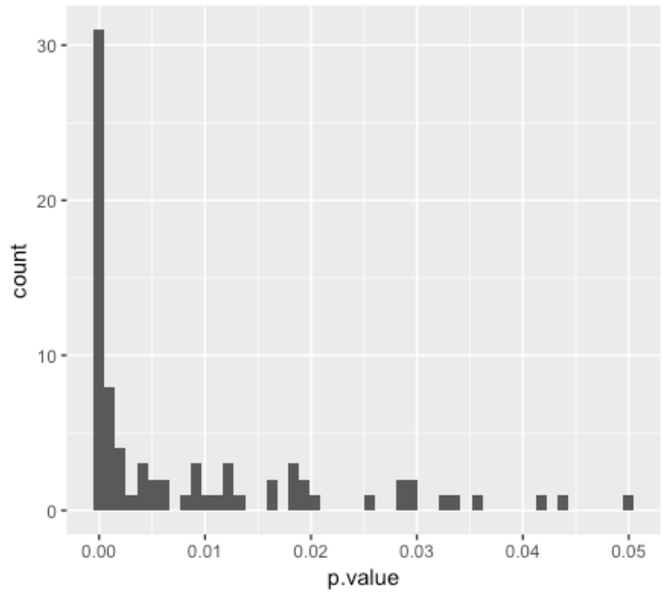
- It takes one parameter input which specifies the sample size you want to simulate.

```
simulate <- function(sample_size) {
  total_samples <- 100
  participant <- rep(1:sample_size, times = 2)
  condition <- c(rep("fast", times = sample_size), rep("slow", times = sample_size))
  all_data <- NULL

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

  all_data$condition <- as.factor(all_data$condition)
  all_data$dv <- as.integer(all_data$dv)
  print(ggplot(all_data, aes(x = condition, y = dv, fill = condition)) + geom_violin() +
  geom_jitter(alpha = .3, width = .05) + guides(fill = FALSE) + facet_wrap(~sample))
  result <- NULL

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

  print(ggplot(result, aes(x = p.value)) + geom_histogram(bins = 50))
  print(ggplot(filter(result, p.value < .05), aes(x = p.value)) + geom_histogram(bins = 50))
  return(count(filter(result, p.value < .05)))

}
```

- Calling it first with `simulate(24)` and then calling it with `simulate(100)` means it runs twice - once for sample size of 24 and once for sample size 100. In the code I ask R to print the number of simulations out of 100 that give us a p-value < .05 and then plot those p-values that are below this critical level.

```
> simulate(24)
# A tibble: 1 x 1
      n
  <int>
1    30

> simulate(100)
# A tibble: 1 x 1
      n
  <int>
1    80
```



- If we wanted to, we could modify our function so that it took 3 parameters (rather than 1) - e.g., sample size, mean (e.g., 1000) of population to sample from, and sd (e.g., 50) of population to sample from - we'd need to change the function definition to:

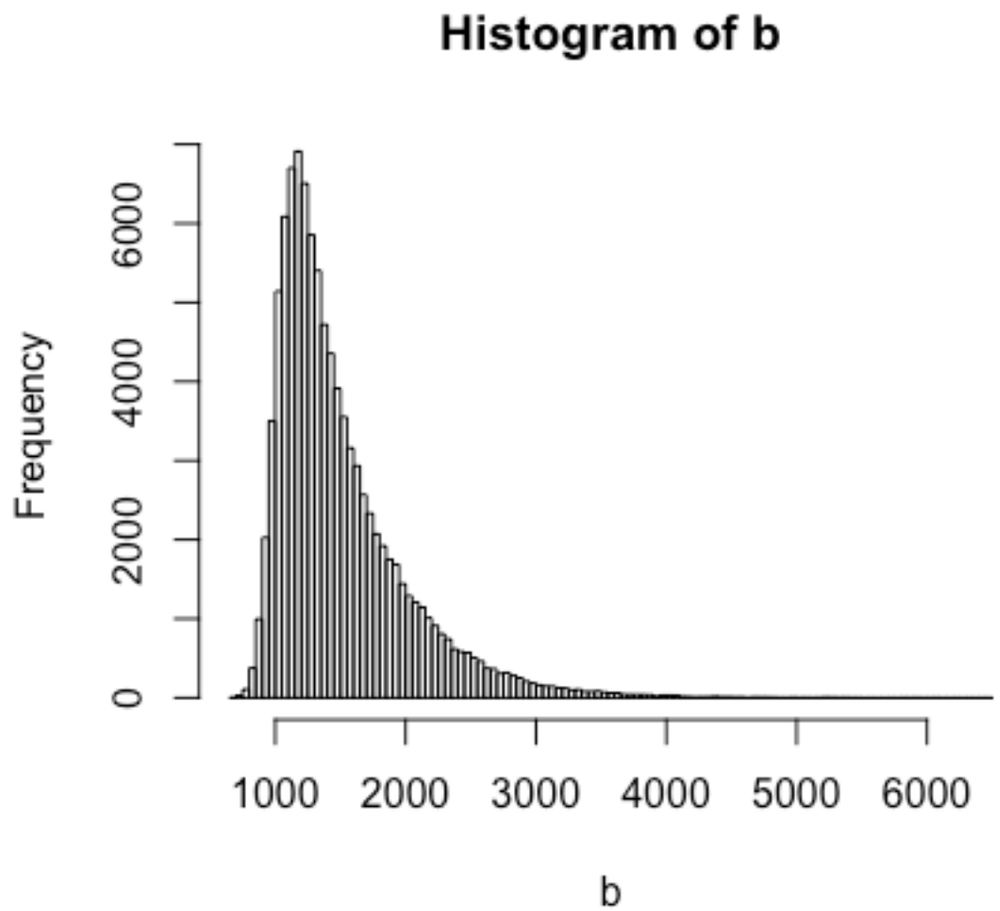
```
> function(sample size, pop mean, pop sd)
```

- and then replace the numbers previously associated with `rnorm()` with the variable names which we now pass to the function. We'd then be able to call our new function like:

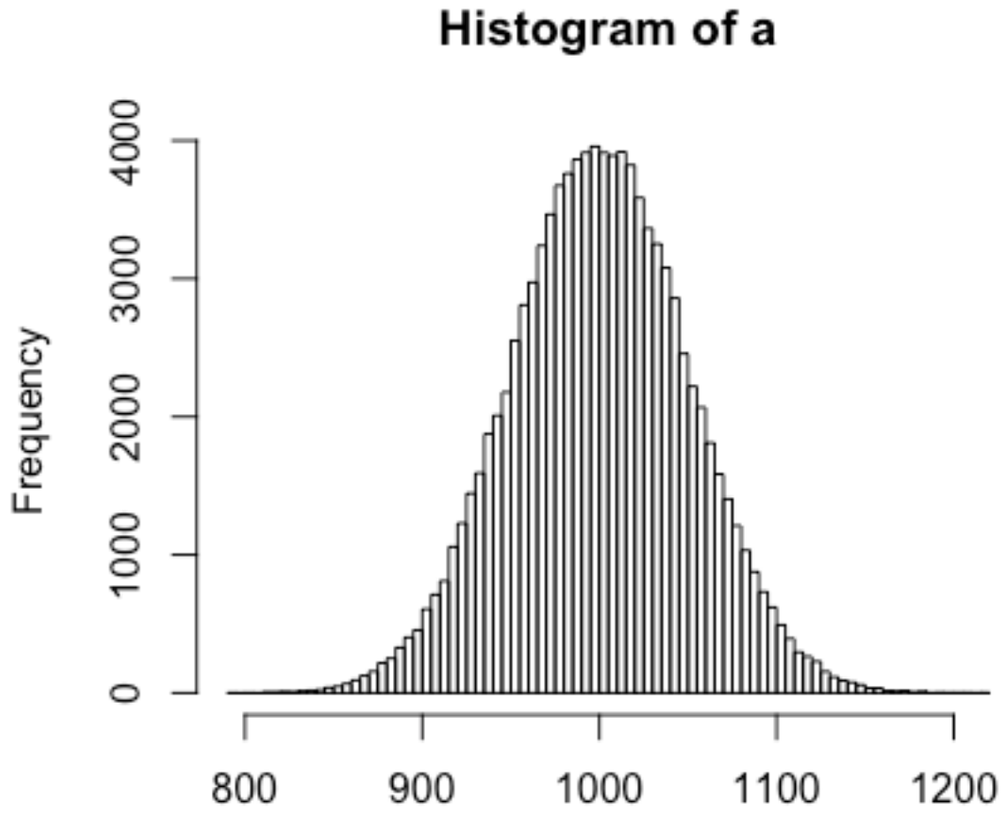
```
> simulate(25, 1000, 50)
```

- So far we've assumed sampling from a normal distribution, but other distributions are available (and may be more appropriate depending on your measure).

- Reaction time data tend to follow the ex-Gaussian distribution - this is a combination of a normal and exponential distribution.

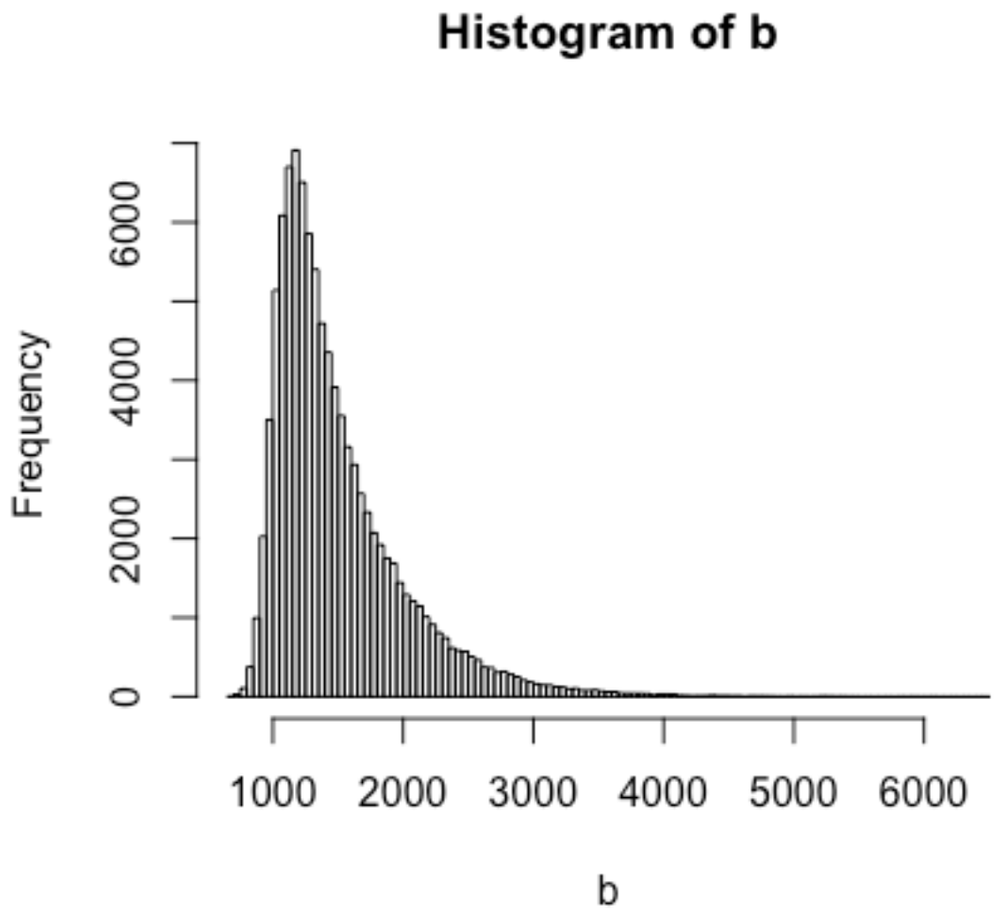



- People tend to have a floor to their reaction times (i.e., they can't react faster than a certain lower limit).




Sampling from the normal distribution

```
a <- rnorm(100000, 1000, 50)
hist(a, breaks = 100)
```




Sampling from the ex-Gaussian distribution

```
b <- rexGAUS(100000, mu = 1000,
  sigma = 100, nu = 500)
hist(b, breaks = 100)
```

mu and sd are the mean and sd of the normal distribution component, nu is the mean of the exponential component.

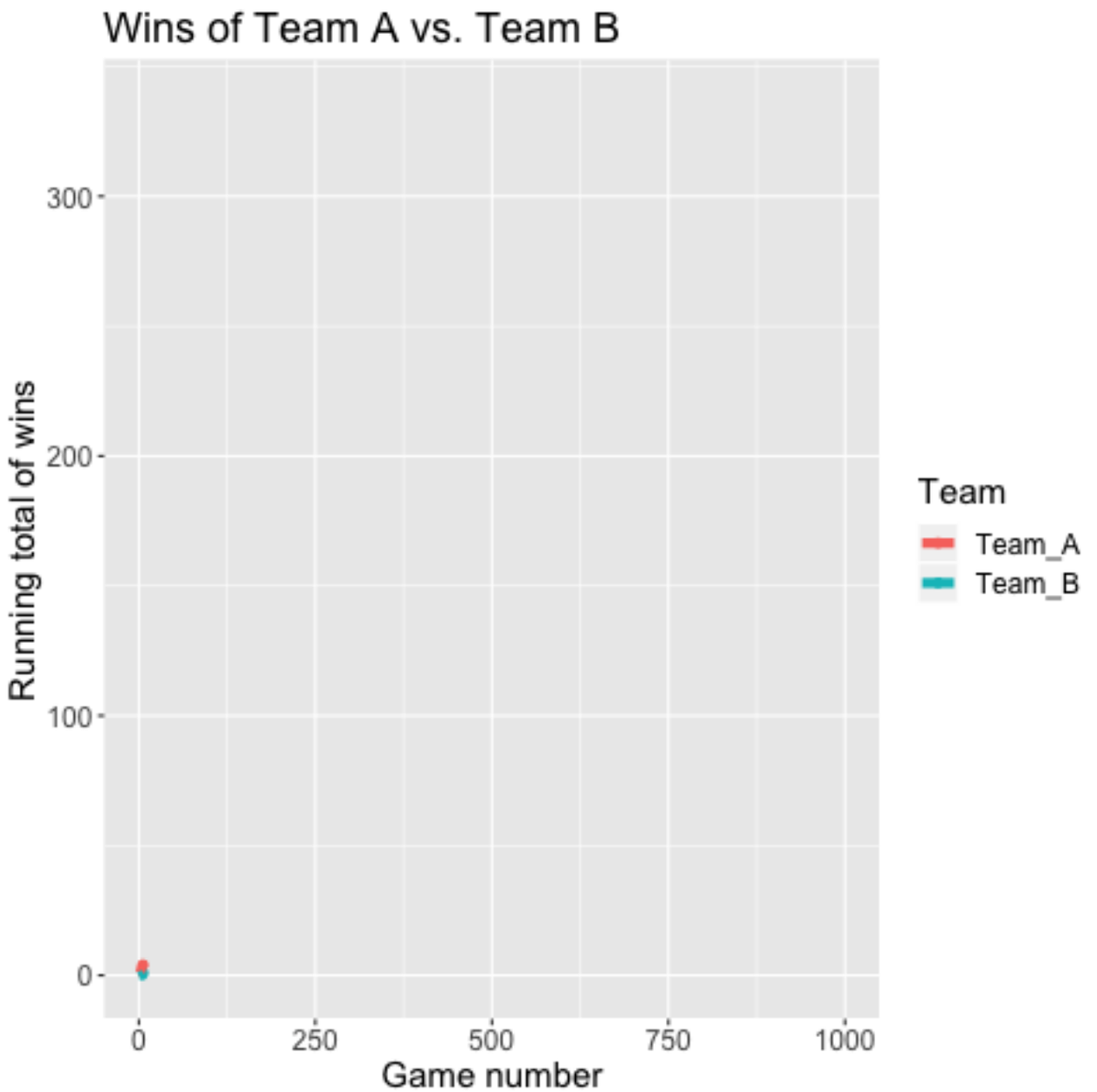
# Hockey game simulation

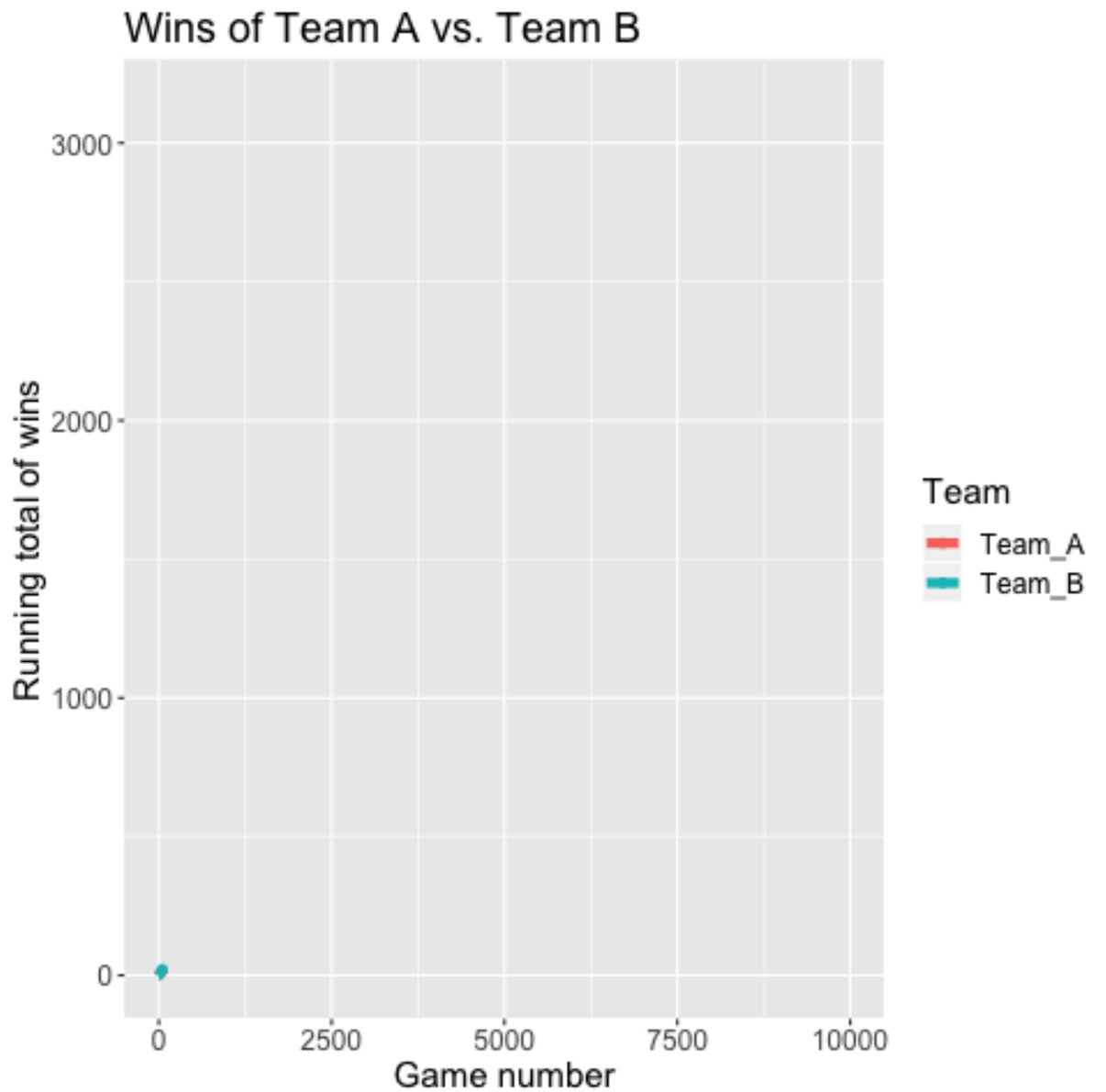
Imagine a hockey game where we know that Team A scores exactly 1 goal for sure and Team B takes 20 shots, each with a 5.5% chance of going in.

Which team would you rather be?

(nothing additional happens if you tie.)

Animation of the first 1,000 games on the left and 10,000 on the right.



# We can code it to simulate the outcomes of 100,000 such games…

```
set.seed(1234)

team_b_goals <- NULL

for(i in 1:100000) {
  score <- sum(sample(c(1, 0), size = 20, replace = TRUE, prob = c(0.055, 1-.055)))
  team_b_goals <- c(team_b_goals, score)}

team_a_goals <- rep(1, 100000)

all_games <- as.tibble(cbind(team_a_goals, team_b_goals))
```

```
> nrow(filter(all_games, team_a_goals > team_b_goals))
[1] 32022
> nrow(filter(all_games, team_a_goals < team_b_goals))
[1] 30337
> nrow(filter(all_games, team_a_goals == team_b_goals))
[1] 37641
```

We see that out of 100,000 simulations, Team A wins on 32,022 occasions.  Team B wins on 30,337 occasions and there are 37,641 ties.

# The gganimate package

- The `gganimate` package needs to be installed separately from the core `tidyverse` packages.

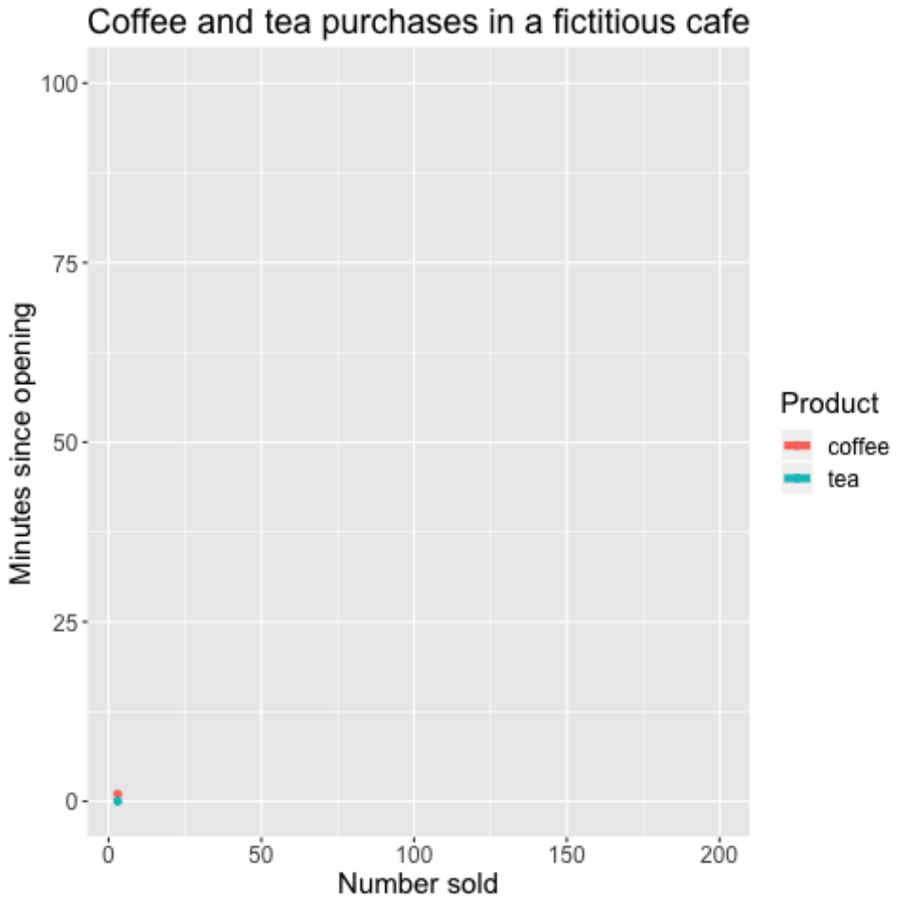
- It follows the Tidyverse philosophy and extends the capabilities of the `ggplot()` function - in may ways it's like adding an extra layer to your plots in the same way you might use `facet_wrap()` but specifying parameters related to your animation frames (and how to transition between those frames).

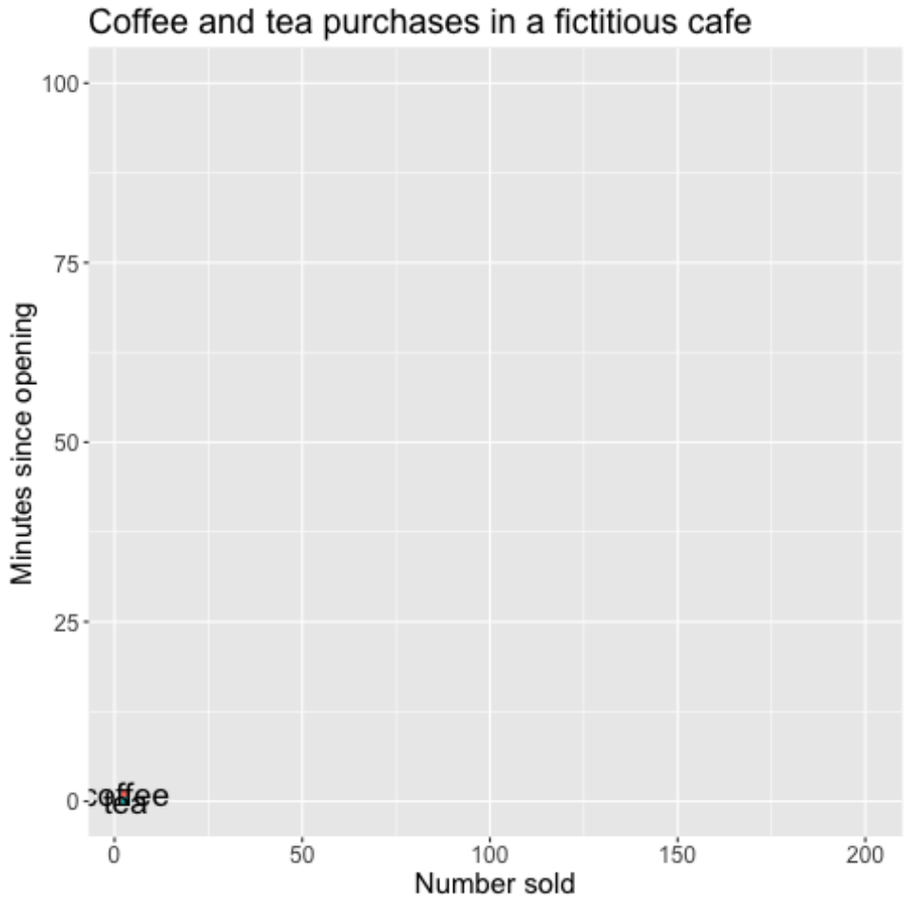
- Let's look at some examples…

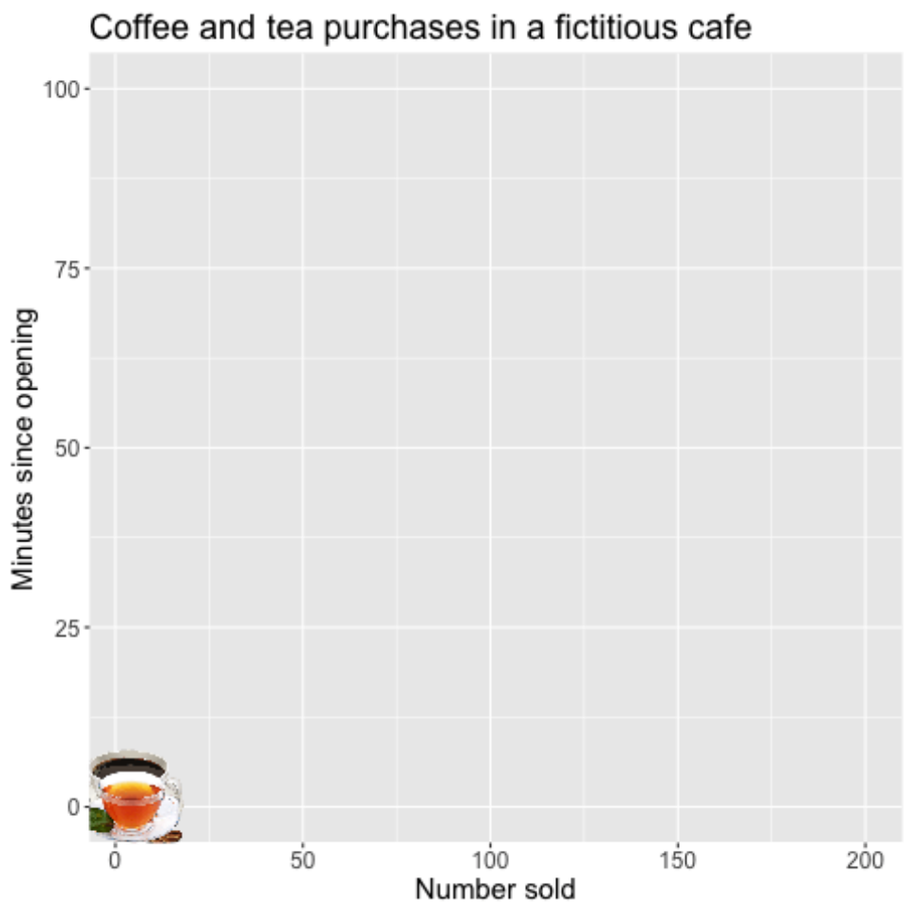
# The data

- Our data set contains information from a cafe about the number of teas and coffees (Product) sold (n_sold) from minutes since opening (Time).

```
> str(data)
Classes ‘tbl_df’, ‘tbl’ and 'data.frame':  200 obs. of  3 variables:
 $ Product: Factor w/ 2 levels "coffee","tea": 1 1 1 1 1 1 1 1 1 1 ...
 $ Time   : int  3 8 9 10 12 13 14 15 20 22 ...
 $ n_sold : int  1 2 3 4 5 6 7 8 9 10 ...

> head(data)
# A tibble: 6 x 3
  Product  Time n_sold
  <fct>   <int>  <int>
1 coffee      3      1
2 coffee      8      2
3 coffee      9      3
4 coffee     10      4
5 coffee     12      5
6 coffee     13      6
```


Coffee and tea purchases in a fictitious cafe
Minutes since opening
Number sold
100
75
50
25
0
0
50
100
150
200
Product
coffee
tea


Coffee and tea purchases in a fictitious cafe
Minutes since opening
Number sold
100
75
50
25
0
0
50
100
150
200
coffee
tea


Coffee and tea purchases in a fictitious cafe
Minutes since opening
Number sold
100
75
50
25
0
0
50
100
150
200

```
library(gganimate)

# Just animated lines
data %>%
  ggplot(aes(x = n_sold, y = Time, group = Product, colour = Product)) +
  geom_point() +
  geom_line(size = 2) +
  coord_flip() +
  transition_reveal(Time) +
  labs(title = "Coffee and tea purchases in a fictitious cafe",
       x = "Minutes since opening",
       y = "Number sold") +
  theme(text = element_text(size = 15))

# With text labels
data %>%
  ggplot(aes(x = n_sold, y = Time, group = Product, colour = Product)) +
  geom_point() +
  geom_line(size = 2) +
  coord_flip() +
  geom_text(aes(label = Product), size = 6, colour = "black") +
  transition_reveal(Time) +
  guides(colour = FALSE) +
  labs(title = "Coffee and tea purchases in a fictitious cafe",
       x = "Minutes since opening",
       y = "Number sold") +
  theme(text = element_text(size = 15))
```

- The `transition_reveal()` function allows you to reveal the data gradually by whatever labelled parameter you specify.

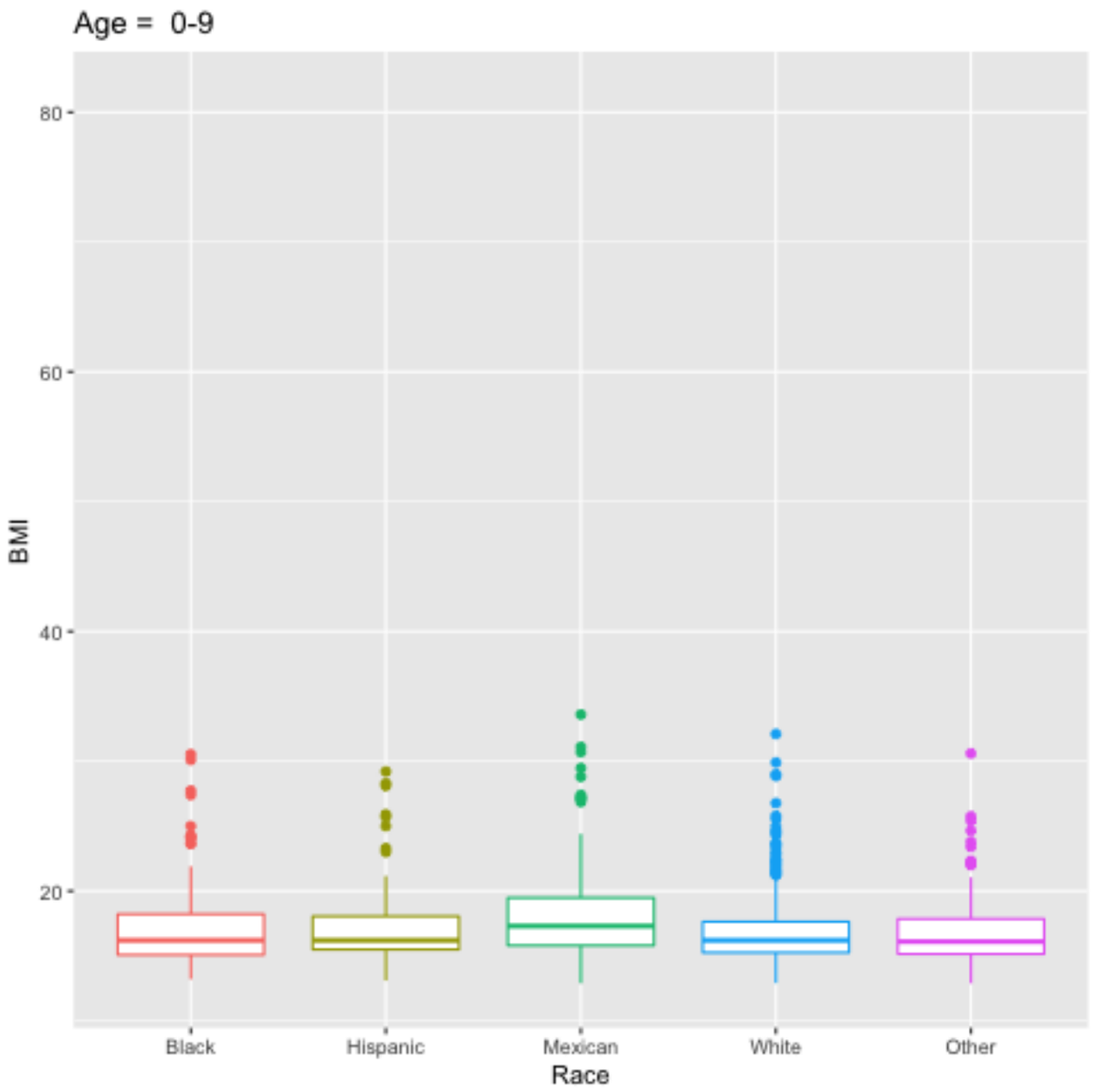
```
library(ggimage)

# Add .png images to animation
data_recoded <- mutate(data, image = recode(Product,
                                            "coffee" = "coffee.png",
                                            "tea" = "tea.png"))

data_recoded %>%
  ggplot(aes(x = n_sold, y = Time, group = image)) +
  coord_flip() +
  geom_line(size = 2, aes(colour = image)) +
  geom_image(aes(image = image), size = .15) +
  transition_reveal(Time) +
  guides(colour = FALSE) +
  labs(title = "Coffee and tea purchases in a fictitious cafe",
       x = "Minutes since opening",
       y = "Number sold") +
  theme(text = element_text(size = 15))
```

Animating aspects of the NHANES dataset:

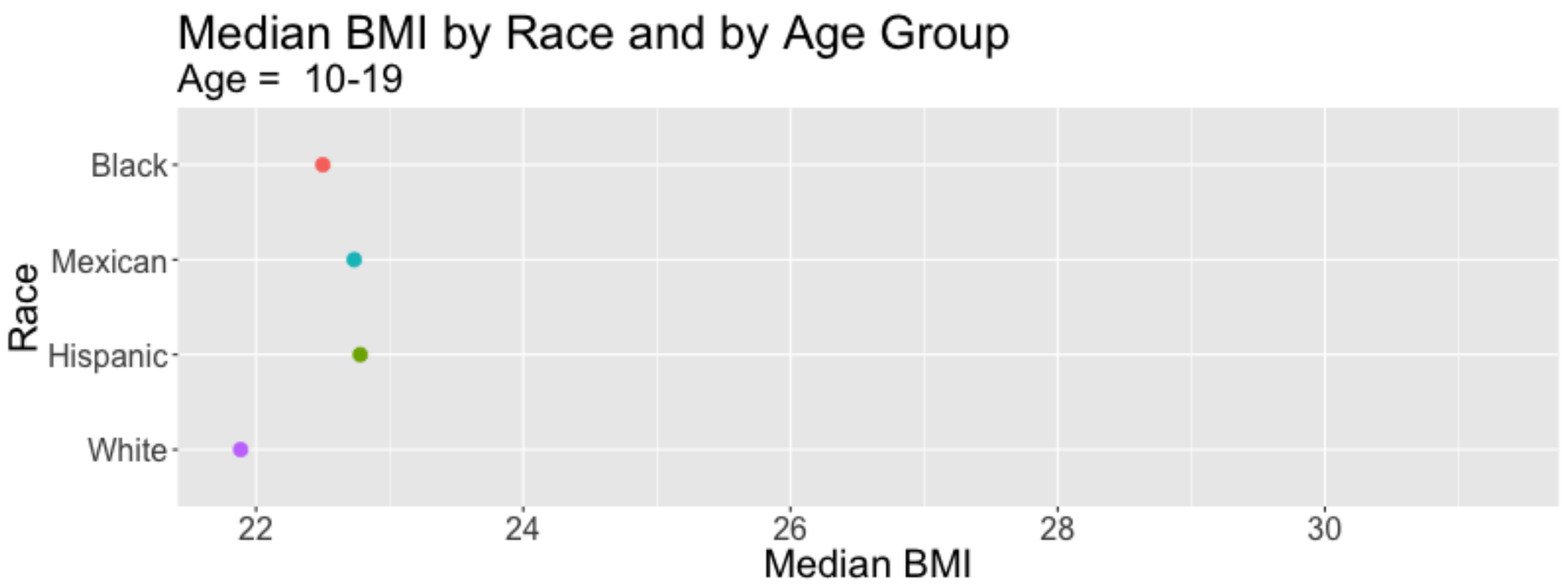
This is survey data collected by the US National Center for Health Statistics (NCHS) which has conducted a series of health and nutrition surveys since the early 1960's. Since 1999 approximately 5,000 individuals of all ages are interviewed in their homes every year and complete the health examination component of the survey. The health examination is conducted in a mobile examination centre (MEC).

```
library(NHANES)

# Boxplot of BMI by Race and AgeDecade
NHANES %>%
  distinct(ID, .keep_all = TRUE) %>%
  ggplot(aes(x = Race1, y = BMI, colour = Race1)) +
  geom_boxplot() +
  guides(colour = FALSE) +
  labs(x = "Race", title = "Age = {closest_state}") +
  transition_states(AgeDecade)
```

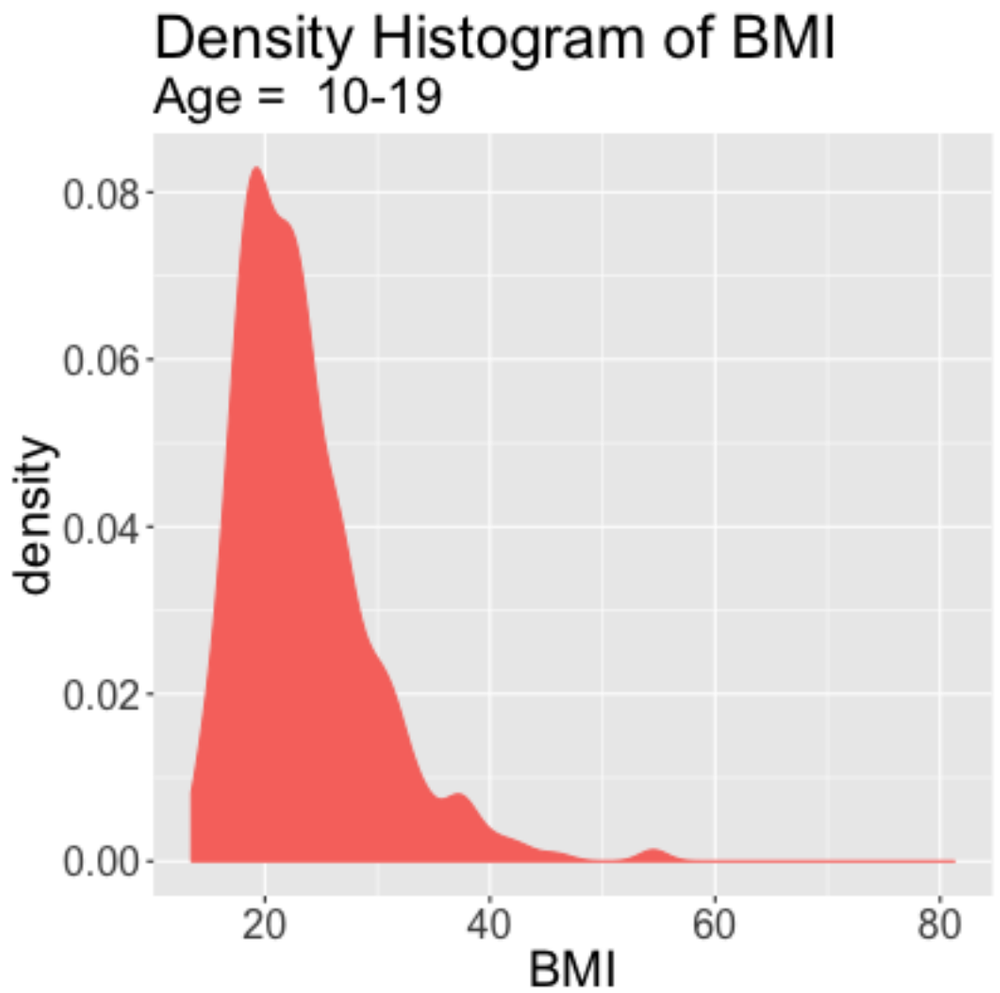
- Think of `transition_states()` like `facet_wrap()` with the transition between each panel animated. The variable `{closest_state}` is available from the `transition_states()` function and can be used outside the function (like I am in the title here).


Age = 0-9
BMI
80
60
40
20
Black
Hispanic
Mexican
White
Other
Race

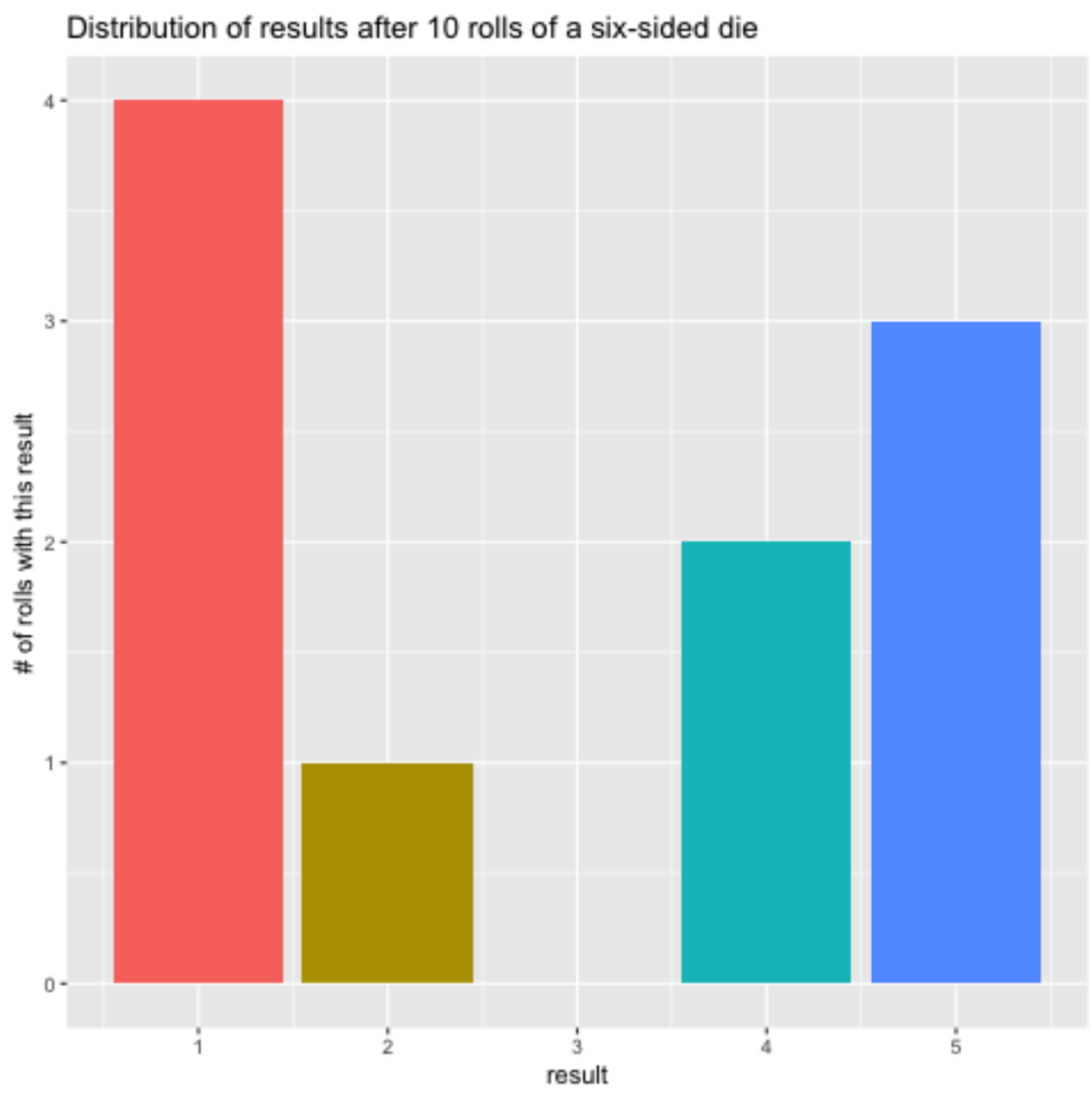
```
NHANES_tidy <- NHANES %>%
  filter(Race1 != "Other") %>%
  filter(as.character(AgeDecade) != " 0-9")

my_plot <- NHANES_tidy %>%
  mutate(AgeDecade = fct_drop(AgeDecade," 0-9")) %>%
  group_by(AgeDecade, Race1) %>%
  summarise(median_BMI = median(BMI, na.rm = TRUE)) %>%
  ggplot(aes(x = median_BMI, y = reorder(Race1, median_BMI), colour = Race1)) +
  geom_point(size = 3) +
  labs(x = "Median BMI", y = "Race", title = "Median BMI by Race and byAge Group",
       subtitle = "Age = {closest_state}") +
  transition_states(AgeDecade) +
  theme(text = element_text(size = 20)) +
  guides(colour = FALSE)

animate(my_plot, height = 300, width = 800)
anim_save("example_plot.gif")
```



# Animated density histogram of BMI



# Simulation of 10,000 dice rolls by @drob




https://twitter.com/drob/status/1100182329350336513

- You can see some really nice slides of examples by the author of the package (Thomas Lin Pedersen, Twitter @thomasp85) from the 2019 RStudio Conference at the following:

https://www.data-imaginist.com/slides/rstudioconf2019/assets/player/keynotedhtmlplayer#11

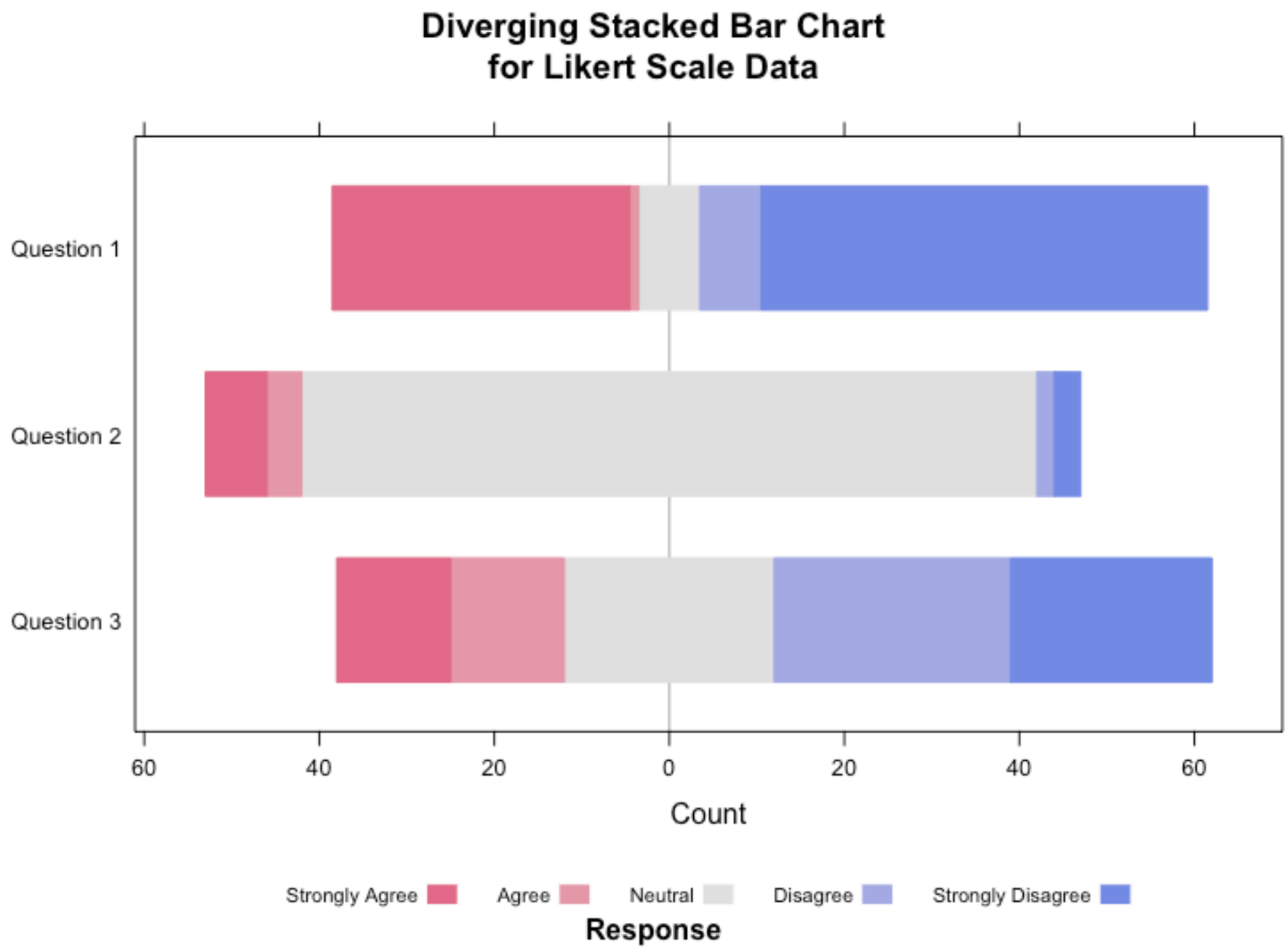
# Visualising Likert Scale Data

Imagine you asked 100 people to respond to three questions using a 5-point Likert scale - their data might look like this:

| | Strongly Agree | Agree | Neutral | Disagree | Strongly Disagree |
|---|---|---|---|---|---|
| Question 1 | 34 | 1 | 7 | 7 | 51 |
| Question 2 | 7 | 4 | 84 | 2 | 3 |
| Question 3 | 13 | 13 | 24 | 27 | 23 |

How might you best communicate the data? It might be misleading to report a measure of central tendency as the mean, median, and mode for Question 2 will all suggest people were Neutral (and ignore those in the tails).

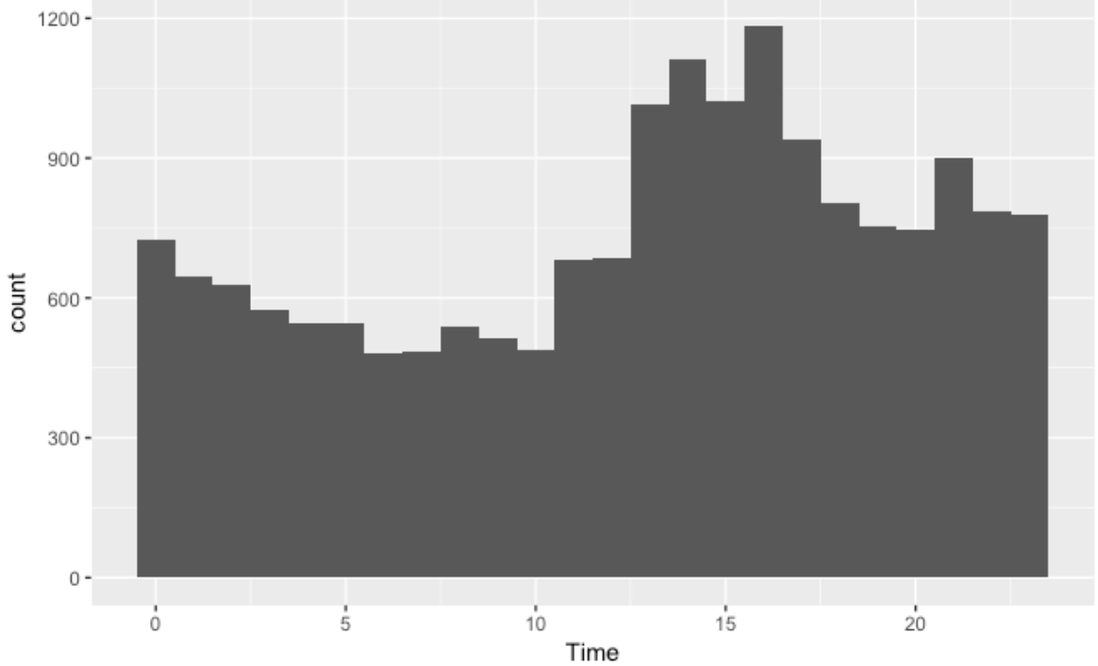
One option is via a diverging stacked bar chart using the `likert()` function in the package `HH`...


Diverging Stacked Bar Chart
for Likert Scale Data
Question 1
Question 2
Question 3
60
40
20
0
20
40
60
Count
Strongly Agree
Agree
Neutral
Disagree
Strongly Disagree
Response

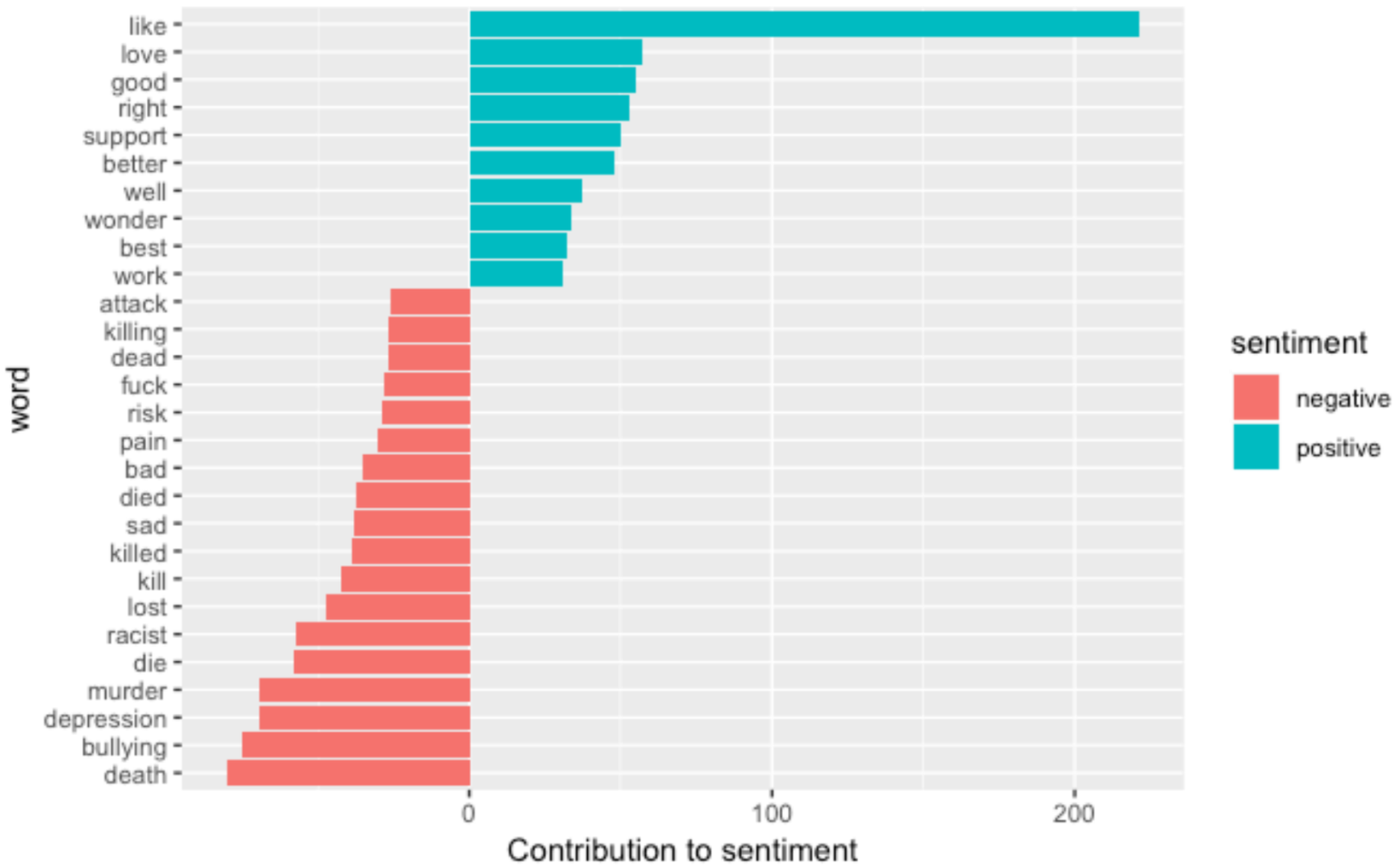
# Visualising Text Data

- We can use the `rtweet` package by Mike Kearney to scrape data from Twitter using the `search_tweets()` function. In this case I'm scraping Twitter for mentions of the word 'suicide' in Tweets, extracting the time the Tweet was created and then plotting on a histogram.

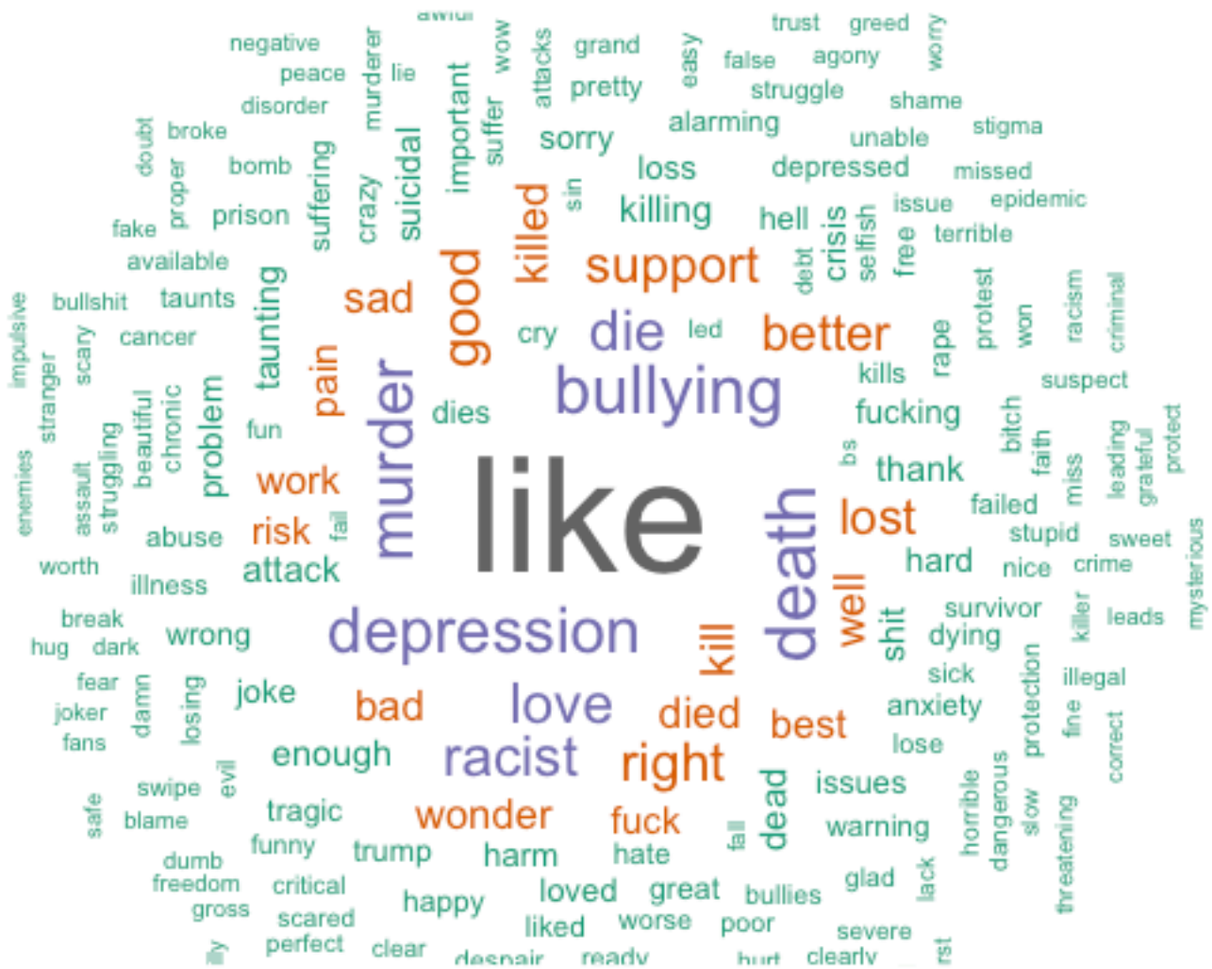
```
tweets <- search_tweets(q = "suicide", n =
1000, include_rts = FALSE, retryonratelimit =
TRUE)

time <- tibble(Time = hour(tweets$created_at))

time %>%
  filter(!is.na(Time)) %>%
  ggplot(aes(x = Time)) +
  geom_histogram()
```



- Now I'm using the `tidytext` package to do a sentiment analysis associated with the words in Tweets mentioning 'suicide' created between midnight and 6AM.

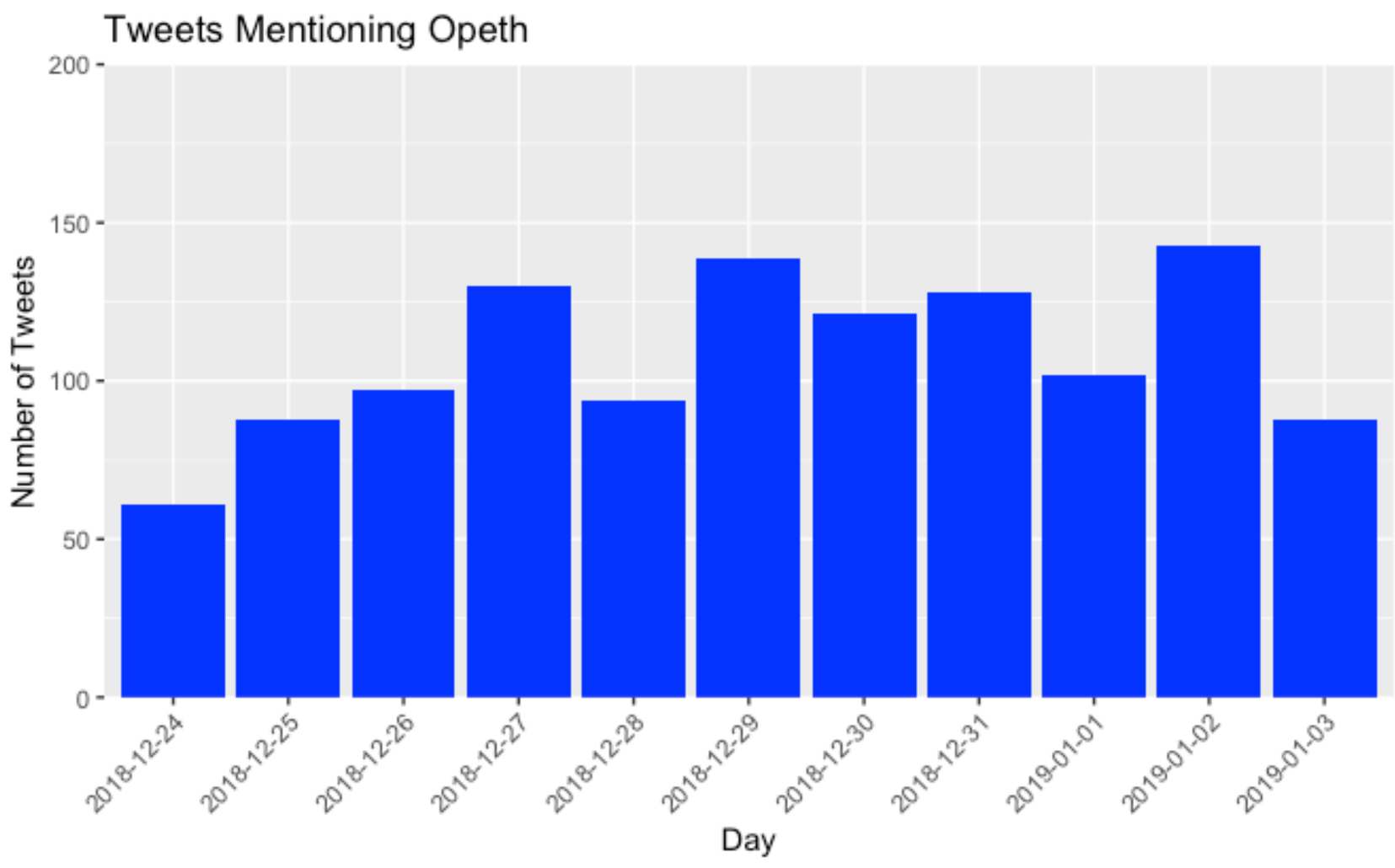


- And visualising the content of the Tweets as a Wordcloud using the `wordcloud` package.



# Searching for mentions of "Opeth"

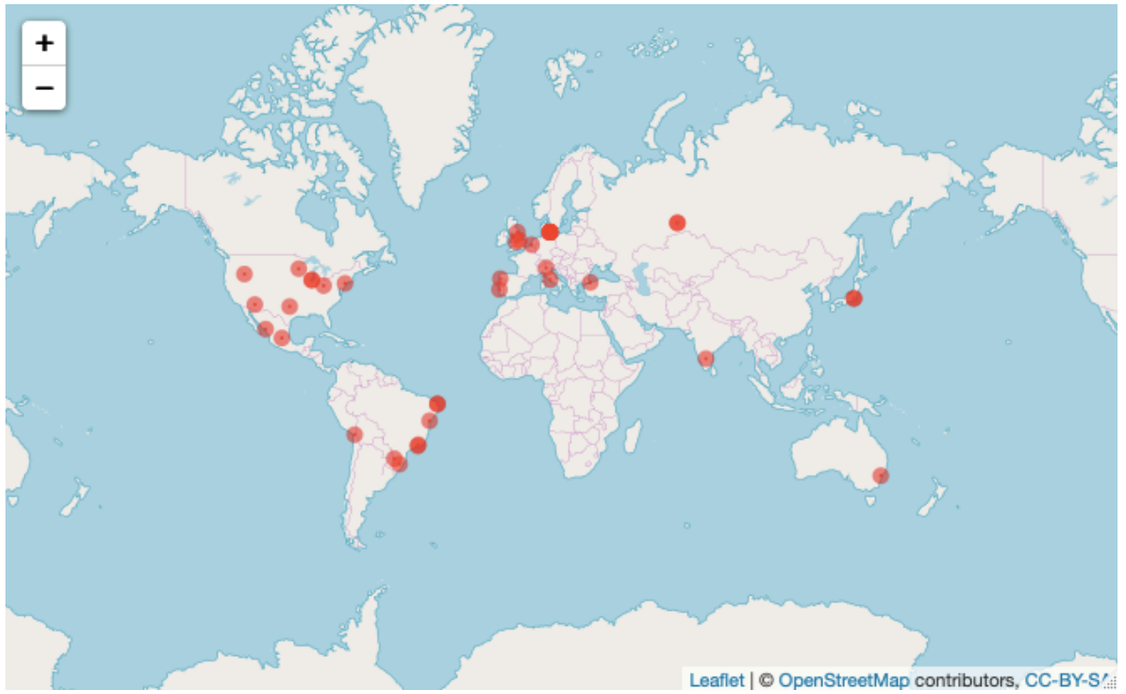
```
library(rtweet)
rt <- search_tweets("Opeth", n = 1000, include_rts = FALSE, retryonratelimit = TRUE)
rt1 <- separate(data = rt, col = created_at, into = c("date", "time"), sep = " ")
rt1 <- rt1[!is.na(rt1$date),]
rt1$date <- as.factor(rt1$date)

ggplot(rt1, aes (x = date)) +
  geom_bar(fill = "blue") +
  scale_y_continuous(expand = c(0,0), limits = c(0,200)) +
  labs(x = "Day", y = "Number of Tweets") +
  ggtitle("Tweets Mentioning Opeth") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



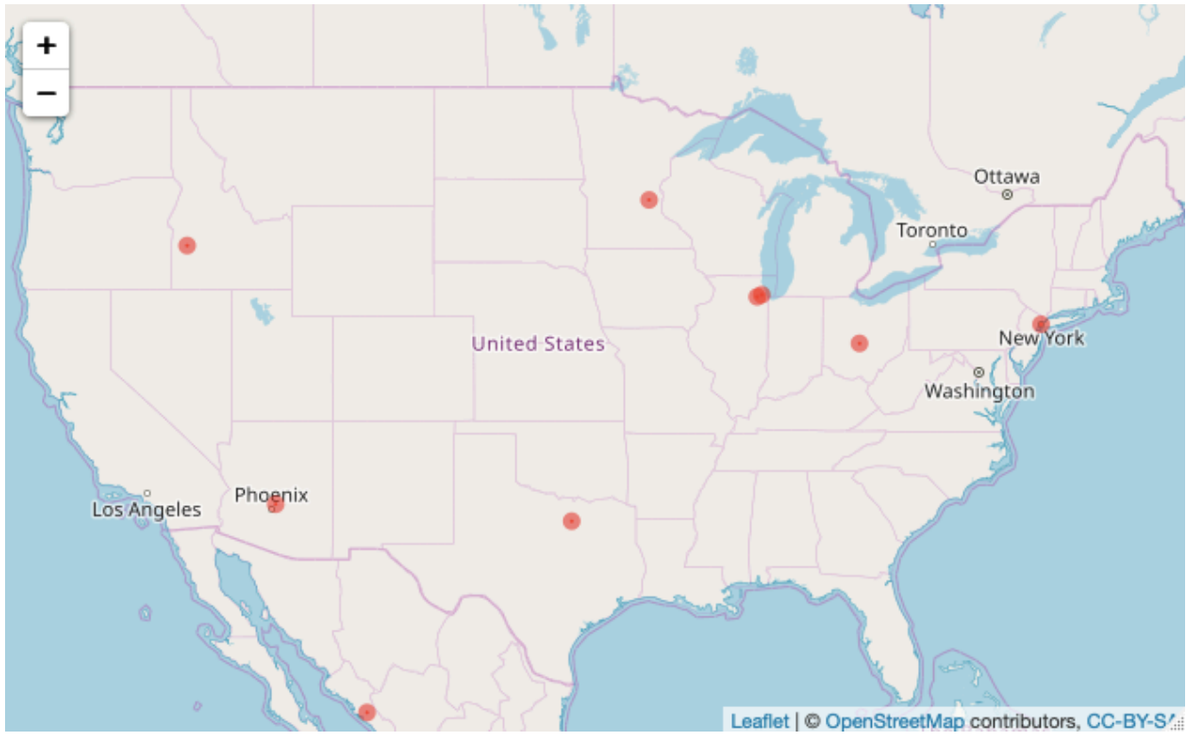
# Geospatial plotting

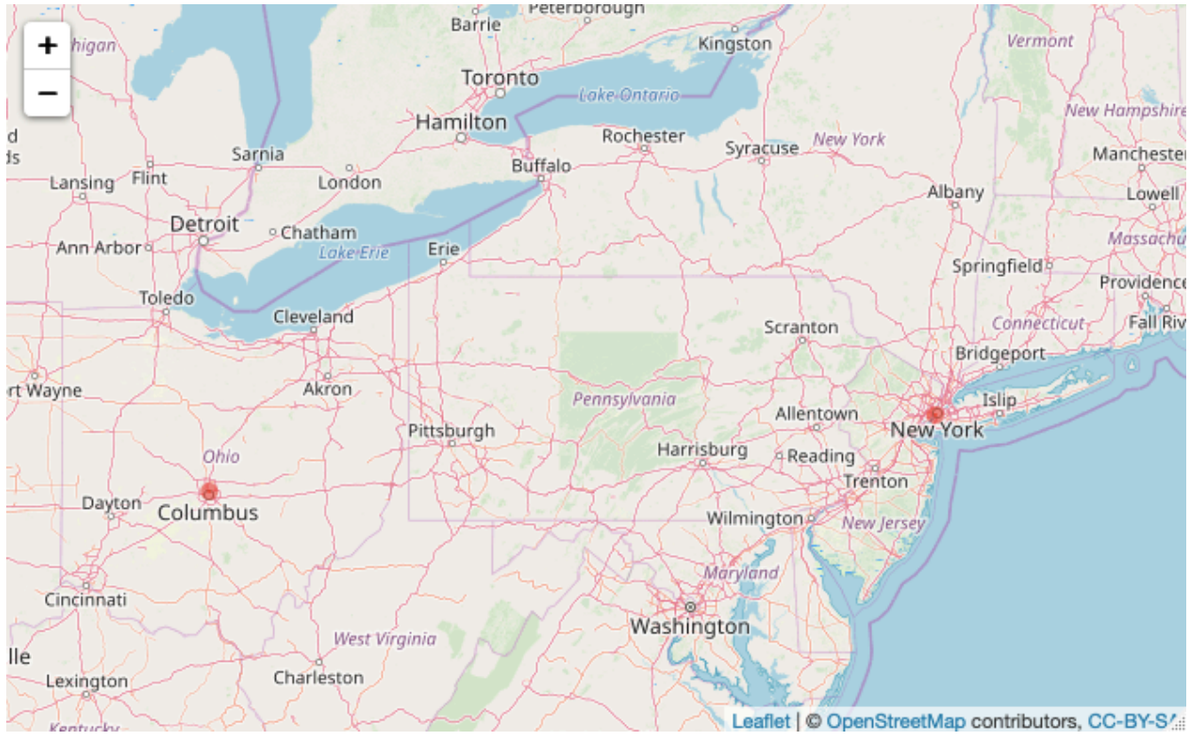
- Some Tweets have associated with them the latitude and longitude of where they were tweeted from - we can use the `leaflet` package to extract these coordinates and plot the location of tweets with geospatial tagging on a map…

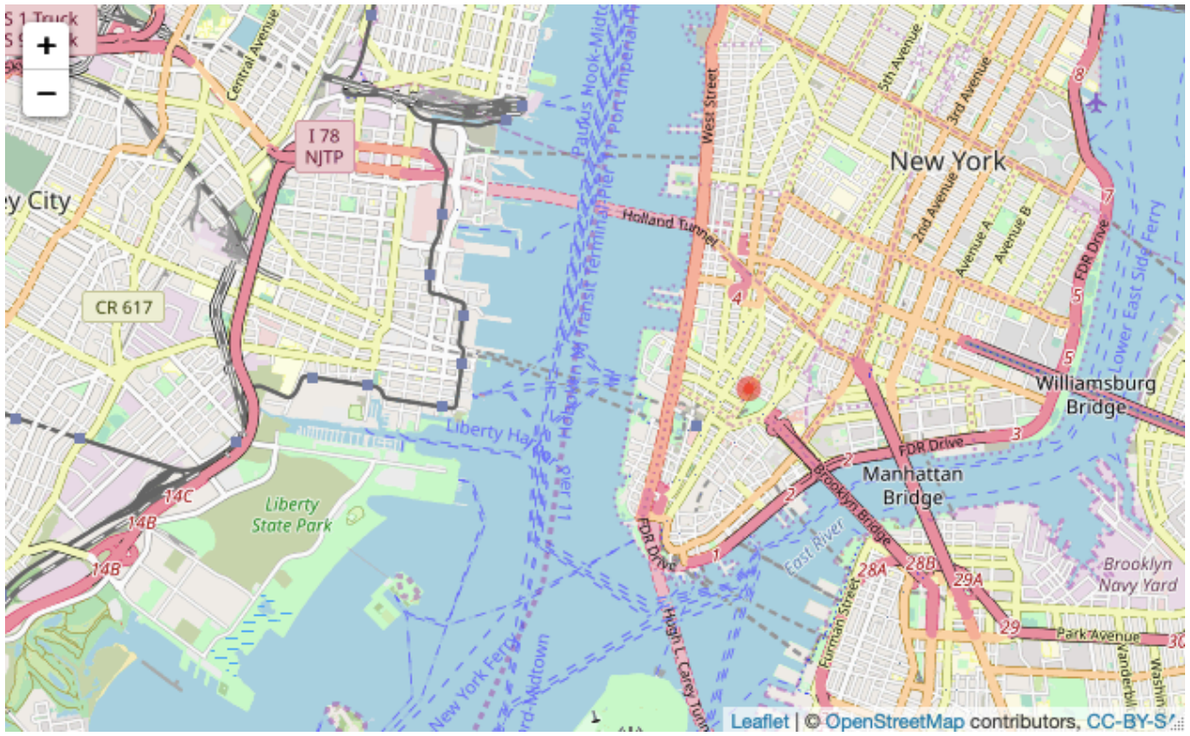
```
library(leaflet)
mymap <- lat_lng(rt)

m <- leaflet(mymap) %>% addTiles()

m %>% addCircles(lng = ~lng, lat =
~lat, weight = 8, radius = 40,
color = "#fb3004", stroke = TRUE,
fillOpacity = 0.8)
```

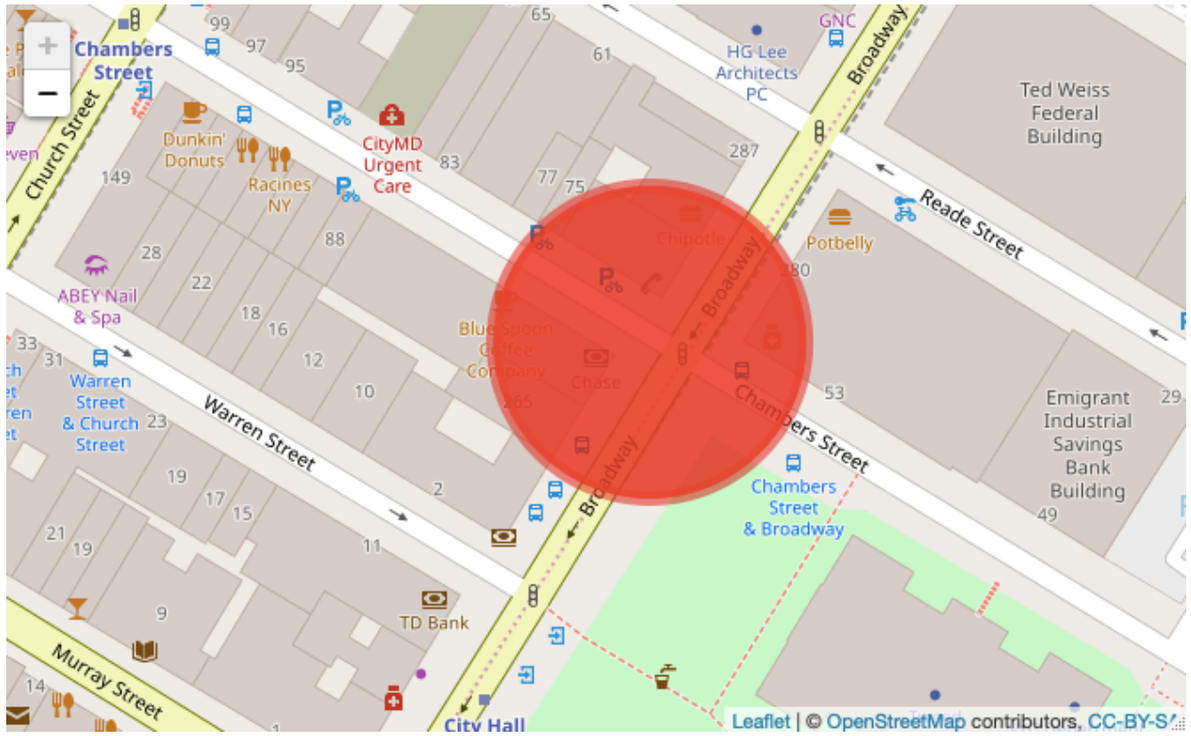


# …and the map is zoomable

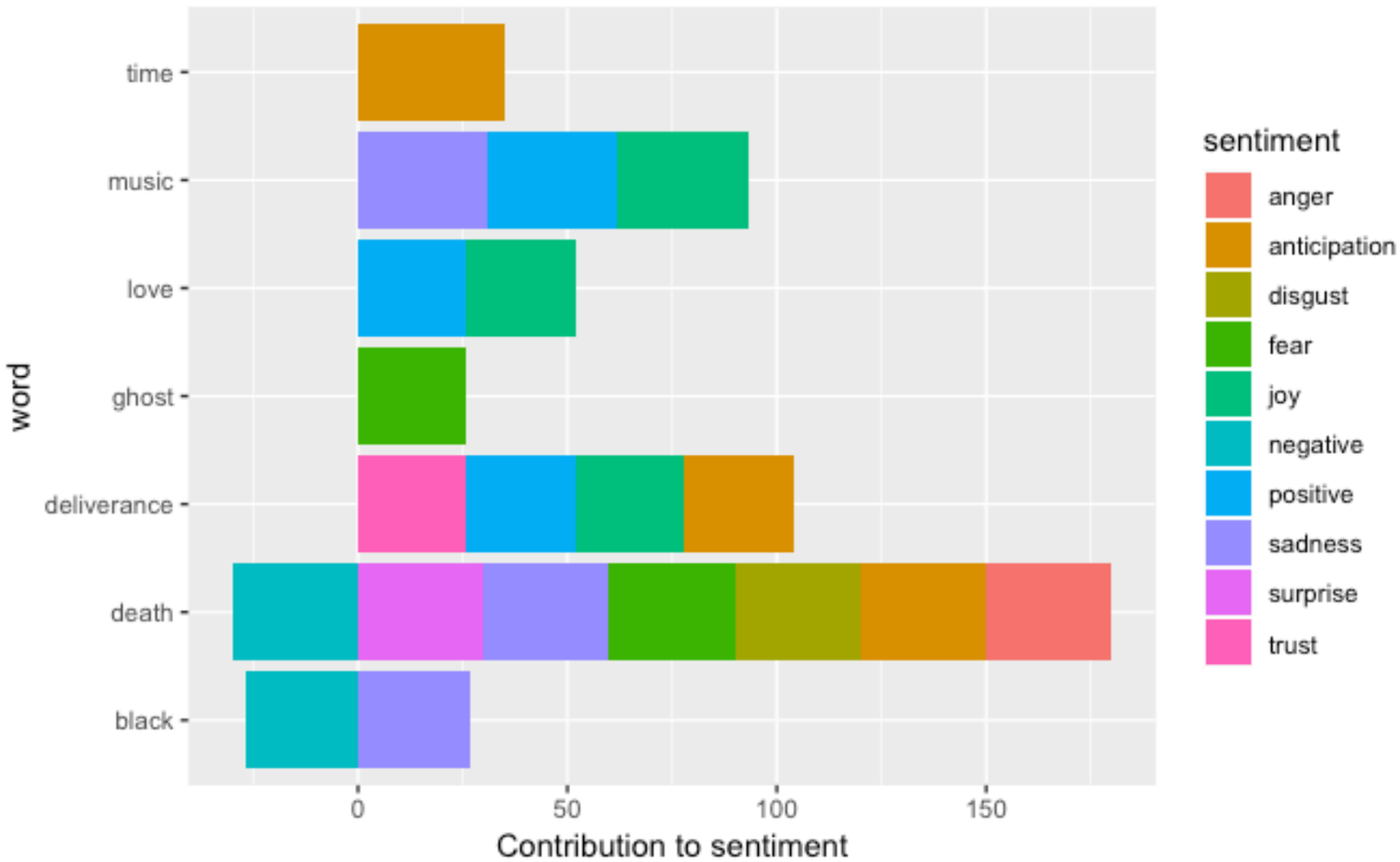


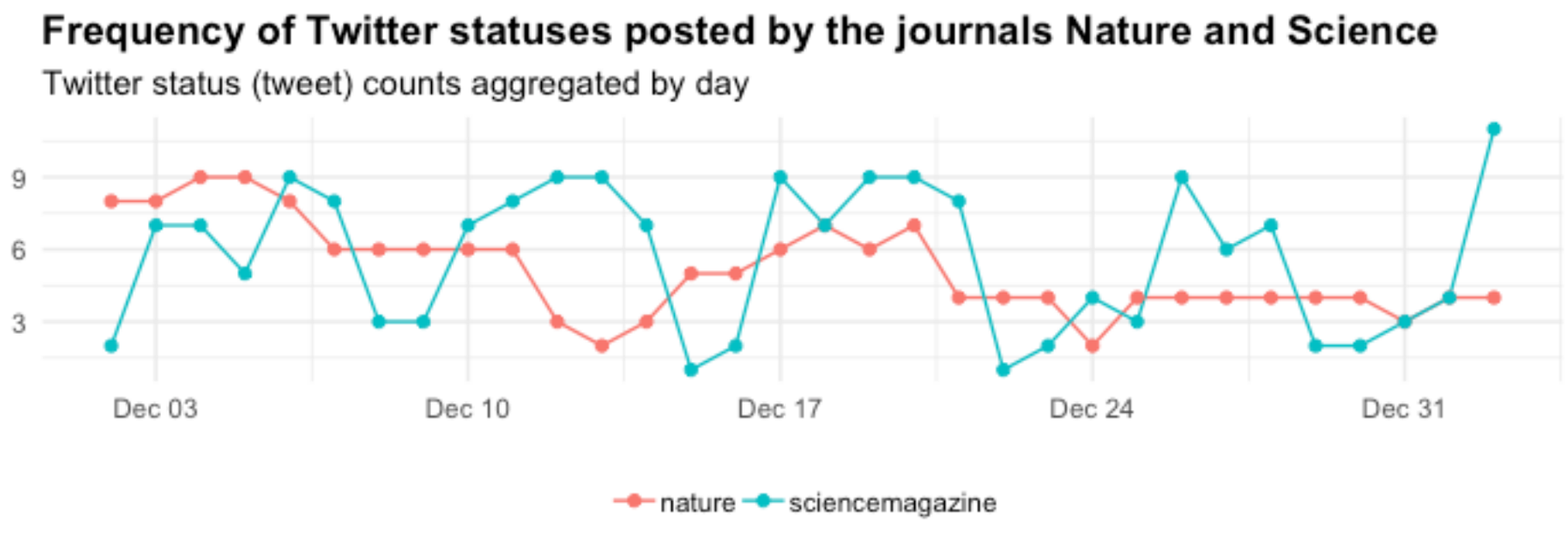



# Sentiment analysis can be more fine-grained than just Positive vs. Negative

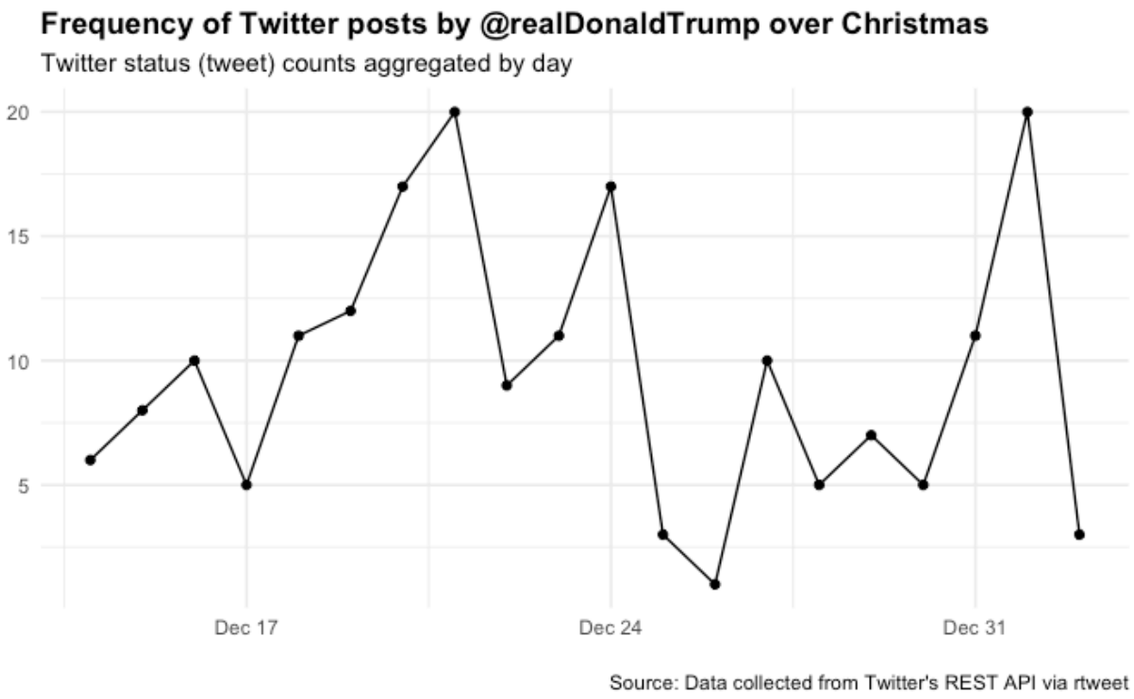


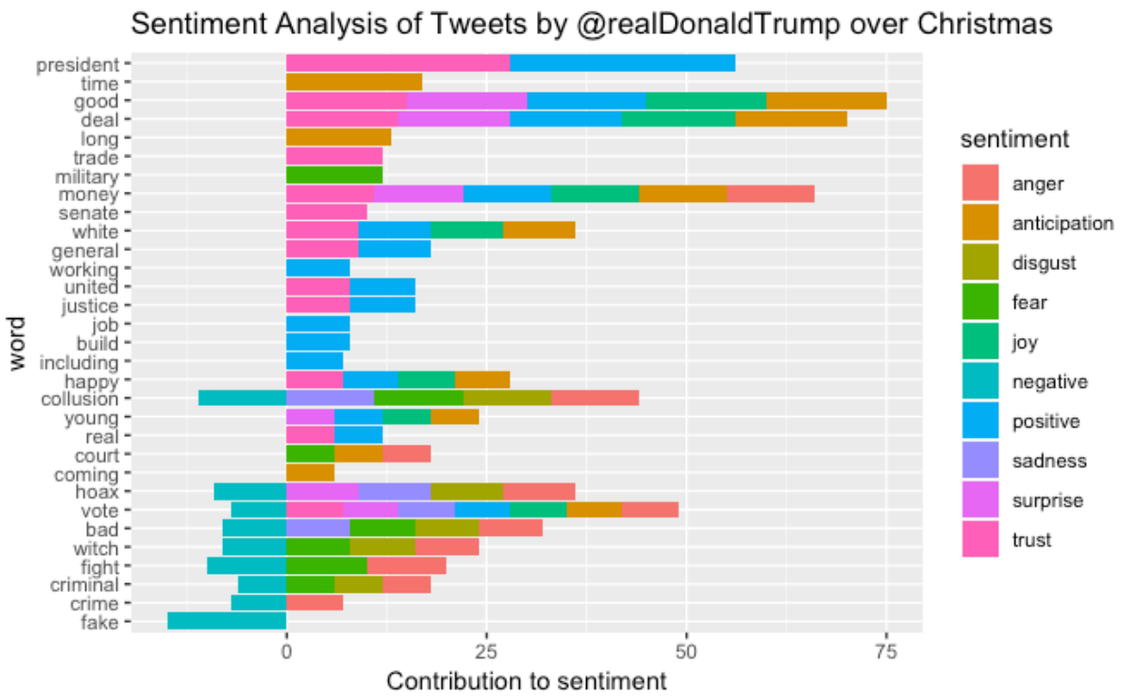
# Collecting the number of Tweets from two timelines

```
tmls <- get_timelines(c("Nature", "sciencemagazine"), n = 1000)
tmls %>%
  filter(created_at > "2018-12-1") %>%
  group_by(screen_name) %>%
  ts_plot("days", trim = 1L) +
  geom_point() +
  theme_minimal() +
  theme(
    legend.title = ggplot2::element_blank(),
    legend.position = "bottom",
    plot.title = ggplot2::element_text(face = "bold")) +
  labs(
    x = NULL, y = NULL,
    title = "Frequency of Twitter statuses posted by the journals Nature and Science",
    subtitle = "Twitter status (tweet) counts aggregated by day",
    caption = "\nSource: Data collected from Twitter's REST API via rtweet"
)
```

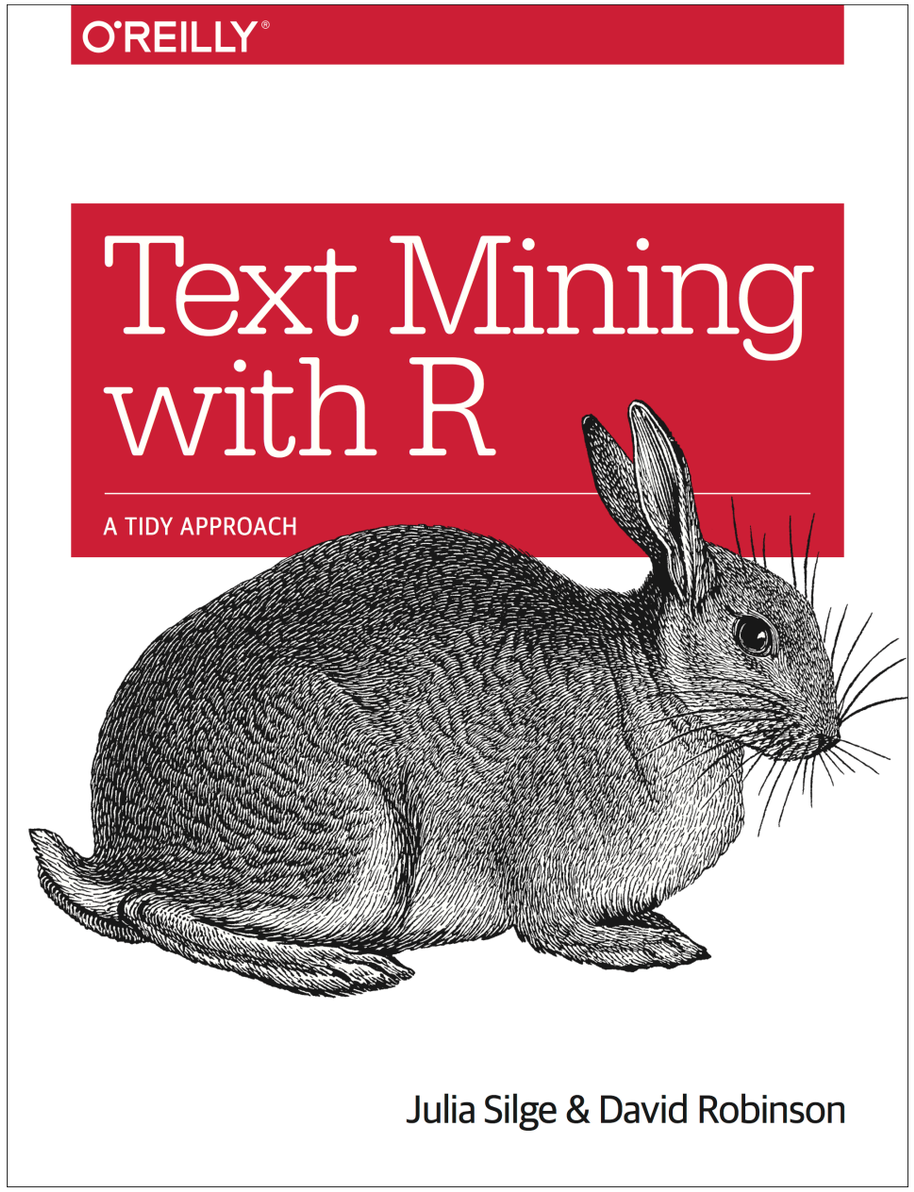


# Collecting Tweets from one timeline




# A good, detailed text mining book for those working on qualitative projects…




https://www.tidytextmining.com

# Reading in texts using the `gutenbergr` package

```
titles <- c("Twenty Thousand Leagues under the Sea", "The War of the Worlds")
books <- gutenberg_works(title %in% titles) %>%
  gutenberg_download(meta_fields = "title")
```

```
> books
# A tibble: 18,609 x 3
   gutenberg_id text                                                    title
          <int> <chr>                                                   <chr>
 1           36 The War of the Worlds                                   The War of the Worlds
 2           36 ""                                                      The War of the Worlds
 3           36 by H. G. Wells [1898]                                   The War of the Worlds
 4           36 ""                                                      The War of the Worlds
 5           36 ""                                                      The War of the Worlds
 6           36 "     But who shall dwell in these worlds if they be"   The War of the Worlds
 7           36 "     inhabited? .  .  .  Are we or they Lords of the"  The War of the Worlds
 8           36 "     World? .  .  .  And how are all things made for man?--" The War of the Worlds
 9           36 "          KEPLER (quoted in The Anatomy of Melancholy)" The War of the Worlds
10           36 ""                                                      The War of the Worlds
# … with 18,599 more rows
```

- In the above code I read in two books using the `gutenbergr` package which reads them from the Project Gutenberg library (containing over 58,000 free books). The object `books` contains the text of both books contained in the “text” column

```
text_waroftheworlds <- books %>%
  filter(title == "The War of the Worlds") %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words)

text_underthesea <- books %>%
  filter(title == "Twenty Thousand Leagues under the Sea") %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words)
```

- I then filter by book title and ‘unnest’ the text of each book so that we have one column in each of two dataframes corresponding to the words in each book…

```
> text_waroftheworlds
# A tibble: 60,513 x 3
   gutenberg_id title                     word
          <int> <chr>                     <chr>
 1           36 The War of the Worlds the
 2           36 The War of the Worlds war
 3           36 The War of the Worlds of
 4           36 The War of the Worlds the
 5           36 The War of the Worlds worlds
 6           36 The War of the Worlds by
 7           36 The War of the Worlds h
 8           36 The War of the Worlds g
 9           36 The War of the Worlds wells
10           36 The War of the Worlds 1898
# … with 60,503 more rows
```

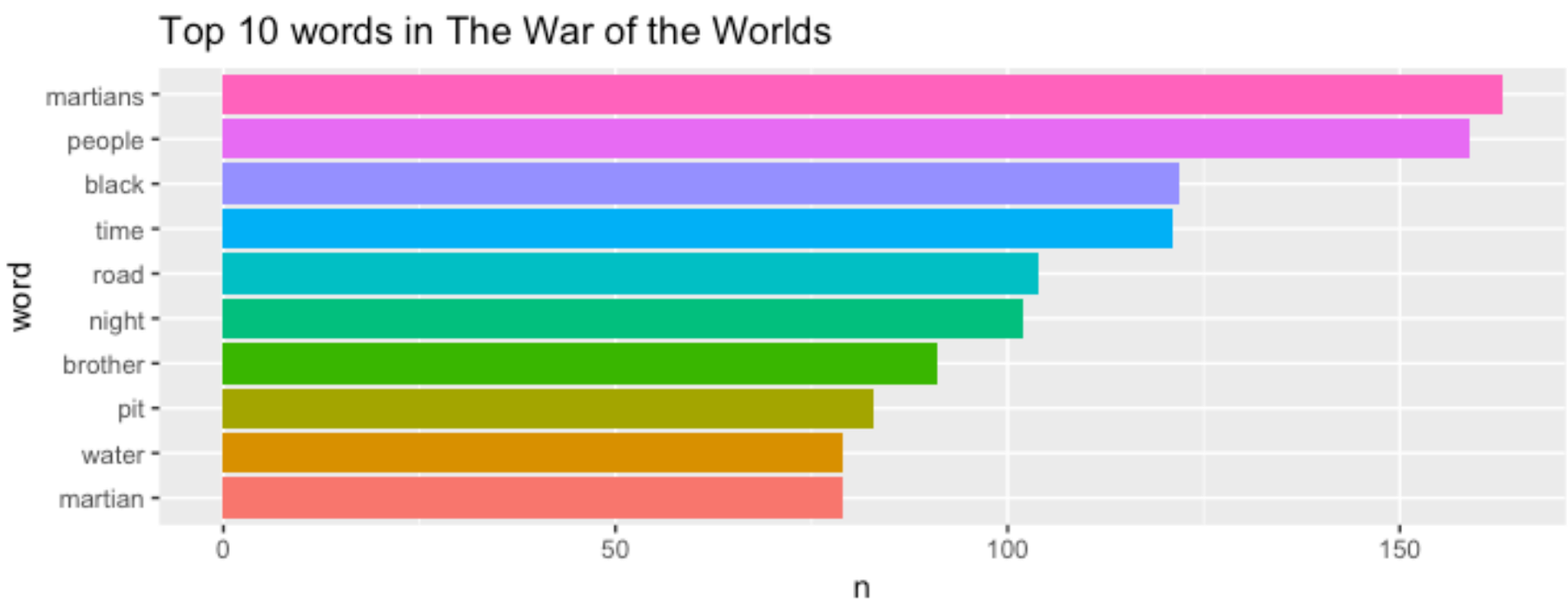
- If I hadn't use the `anti_join(stopwords)` call I would end up with a data frame like the above containing lots of common function words.

- But by adding the `anti_join(stopwords)` line to my code I exclude all stop words (the common function words) and end up with:

```
> text_waroftheworlds
# A tibble: 22,583 x 3
   gutenberg_id title                     word
          <int> <chr>                     <chr>
 1           36 The War of the Worlds war
 2           36 The War of the Worlds worlds
 3           36 The War of the Worlds 1898
 4           36 The War of the Worlds dwell
 5           36 The War of the Worlds worlds
 6           36 The War of the Worlds inhabited
 7           36 The War of the Worlds lords
 8           36 The War of the Worlds world
 9           36 The War of the Worlds kepler
10           36 The War of the Worlds quoted
# … with 22,573 more rows
```

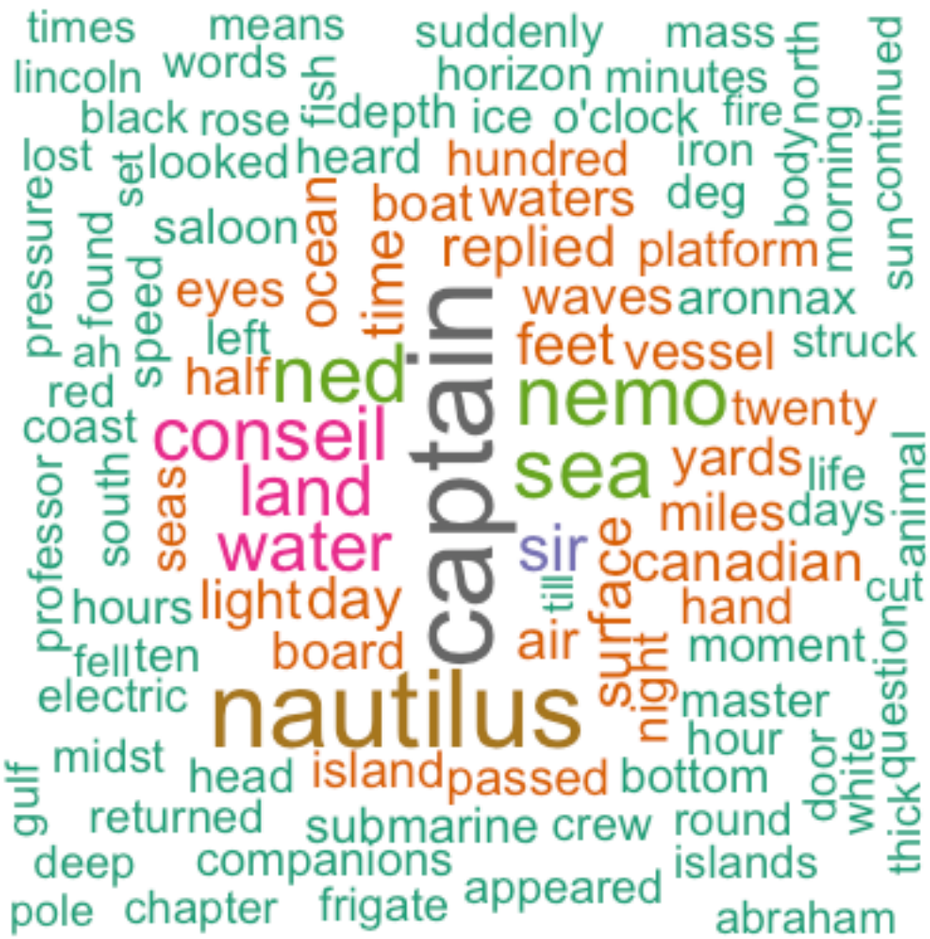
- I can the plot the top 10 most common words in The War of the Worlds with this code:

```
text_waroftheworlds %>%
  count(word) %>%
  top_n(10) %>%
  mutate(word = reorder(word, n)) %>%
  ggplot(aes(x = word, y = n, fill = word)) +
  geom_col() +
  coord_flip() +
  guides(fill = FALSE) +
  labs(title = "Top 10 word in The War of the Worlds")
```



- Plotting a wordcloud based on the words in 20,000 Leagues Under the Sea.

```
text_underthesea_count <- text_underthesea %>%
  count(word) %>%
  top_n(200)

wordcloud(words = text_underthesea_count$word,
          freq = text_underthesea_count$n,
          min.freq = 1,
          scale = c(3, 1),
          max.words = 200,
          random.order = FALSE,
          rot.per = 0.35,
          colors = brewer.pal(8, "Dark2"))
```



# BBC-style Visualisations

- The BBC have published their R graphics cookbook for generating data visualisations following the BBC style guide.

- The cookbook can be found here:

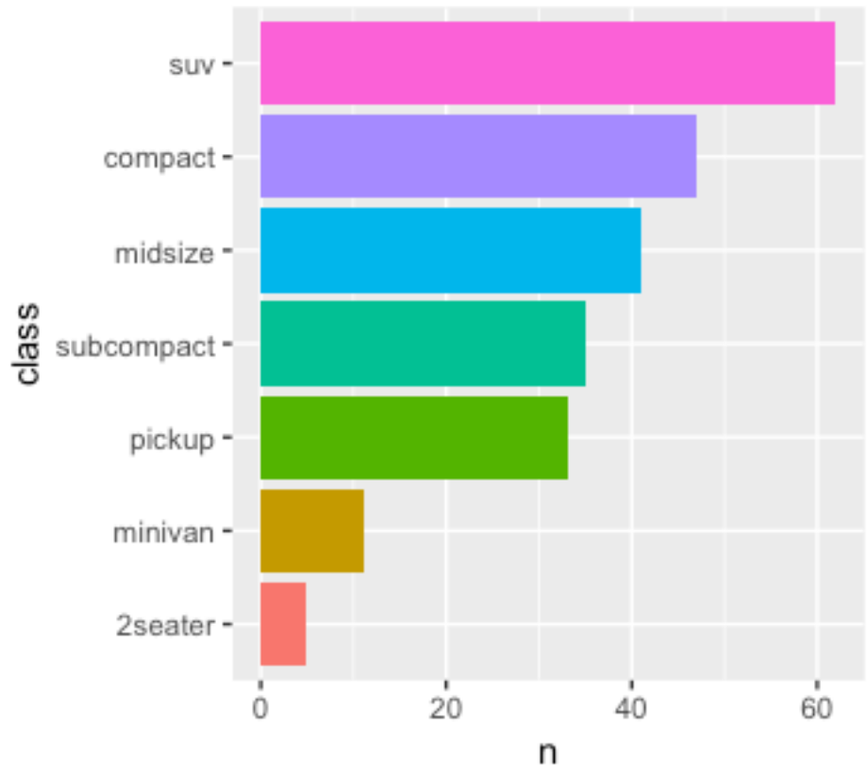
  `https://bbc.github.io/rcookbook/#how does the bbplot package work`

- With the `bbplot` package containing the functions used to generate BBC-style visualisations available on GitHub:

  `https://github.com/bbc/bbplot`

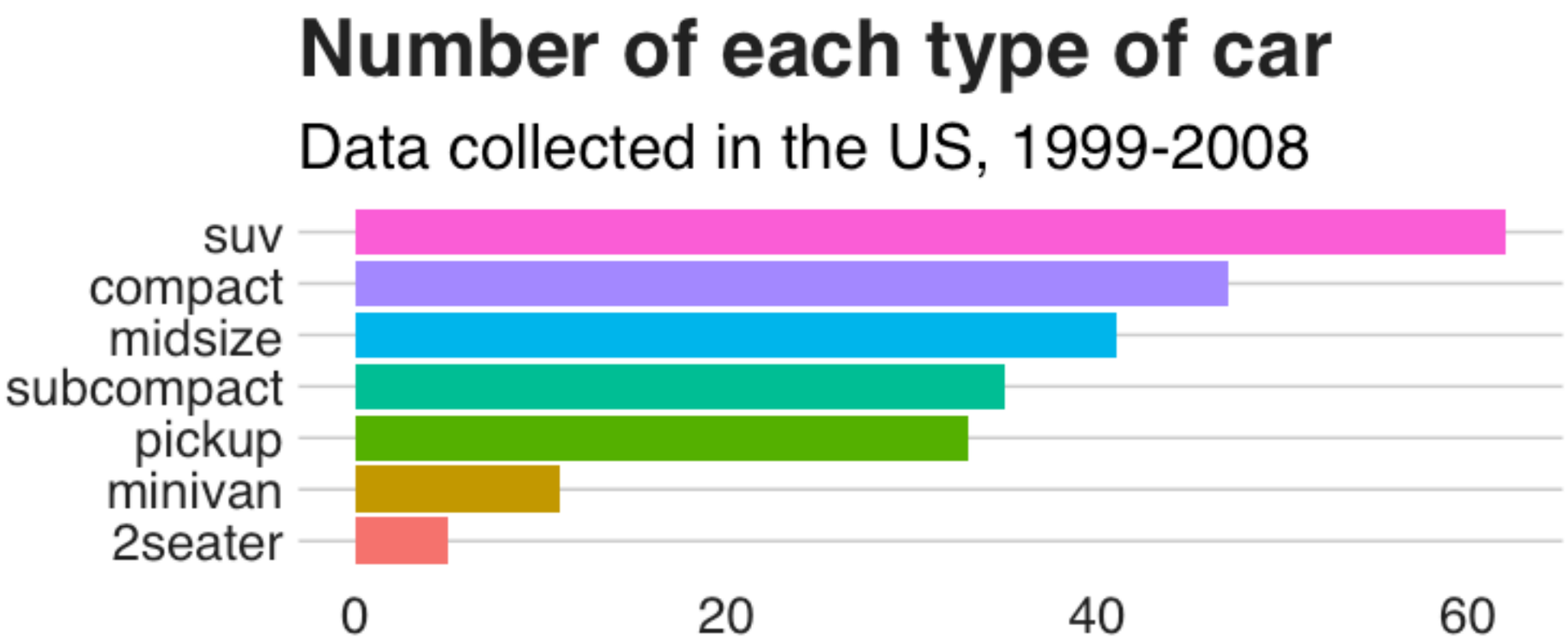
- Let's start with a basic plot - we're going to use the built-in `mpg` dataset.

```
devtools::install_github('bbc/bbplot')

library(tidyverse)
library(bbplot)

my_plot <- mpg %>%
  count(class) %>%
  mutate(class = fct_reorder(class, n)) %>%
  ggplot(aes(x = class, y = n, fill = class)) +
  geom_col() +
  coord_flip() +
  guides(fill = FALSE)

my_plot
```



- We can add an extra parameter to our ggplot code to set the BBC theme using `bbc_style()`:

```
my_plot <- mpg %>%
  count(class) %>%
  mutate(class = fct_reorder(class, n)) %>%
  ggplot(aes(x = class, y = n, fill = class)) +
  geom_col() +
  coord_flip() +
  guides(fill = FALSE)  +
  bbc_style()+
  labs(title="Number of each type of car",
       subtitle = "Data collected in the US, 1999-2008”)
```



# Version Control using git

- Have you ever added some code, or deleted something, or used a weird (accidental) keyboard shortcut and your once working script is now kaput?

- You need Version Control!

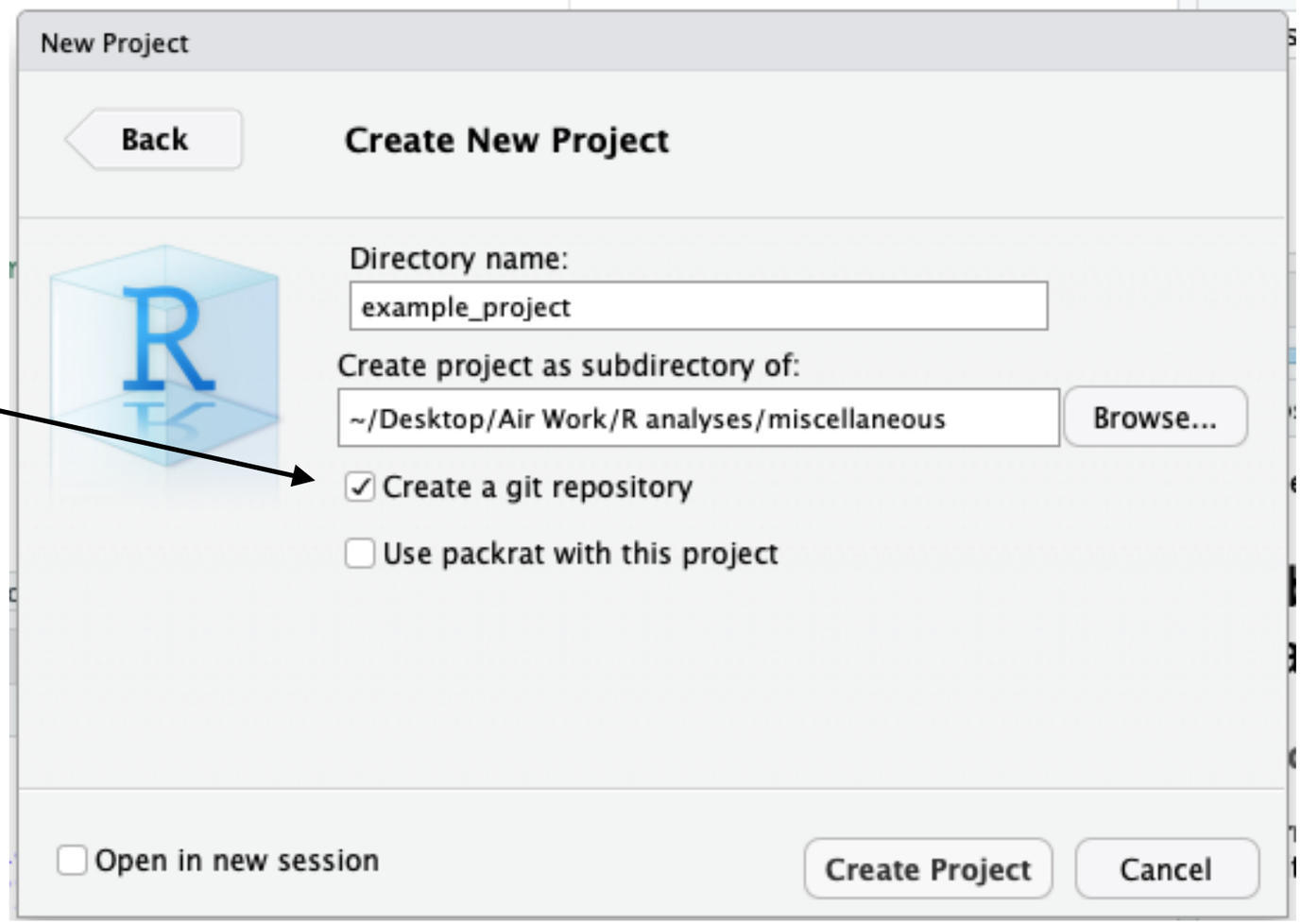
- Version control allows you to revert your script to a previous state (i.e., before you messed it up!)

- In RStudio, you can enable Version Control in a couple of ways - we're going to look at using git.

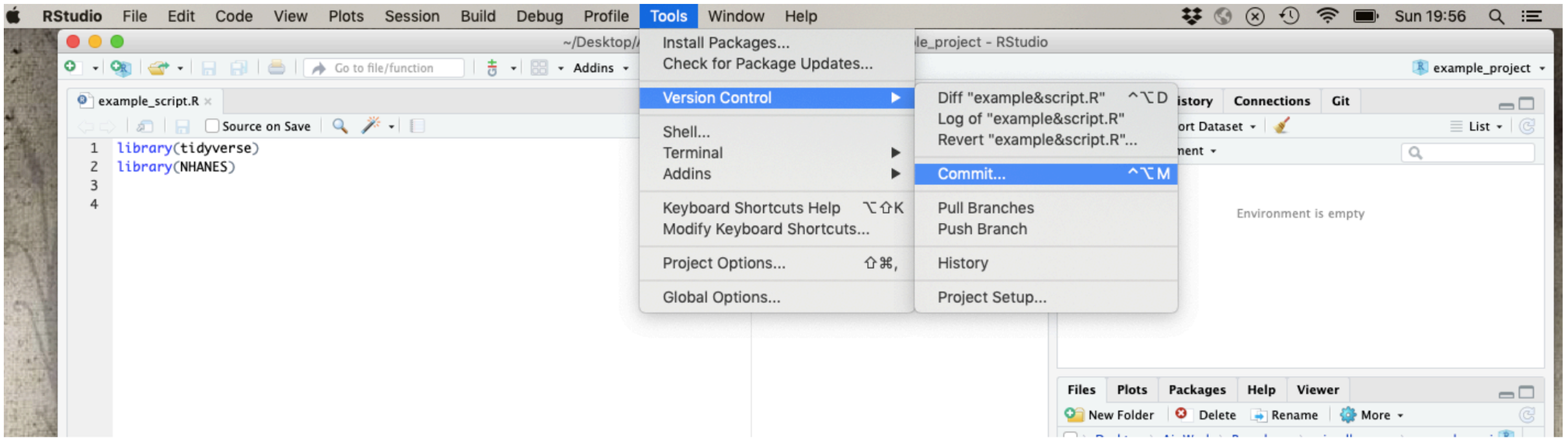
- The first thing you need to do is to set up git on your computer (may not work on cluster PCs):

`https://git-scm.com/downloads`

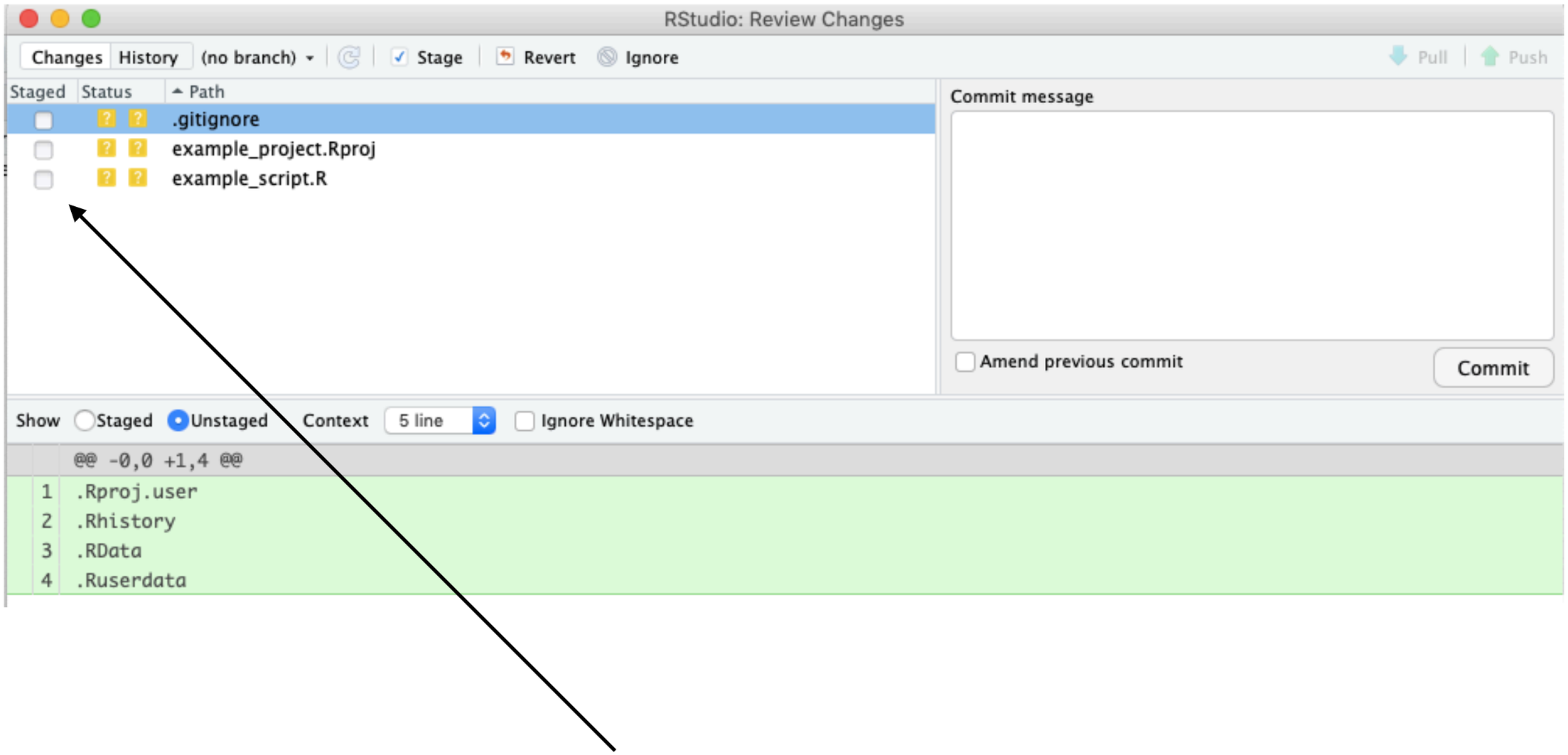
- Then when you create a new project, tick the “Create a git repository” box



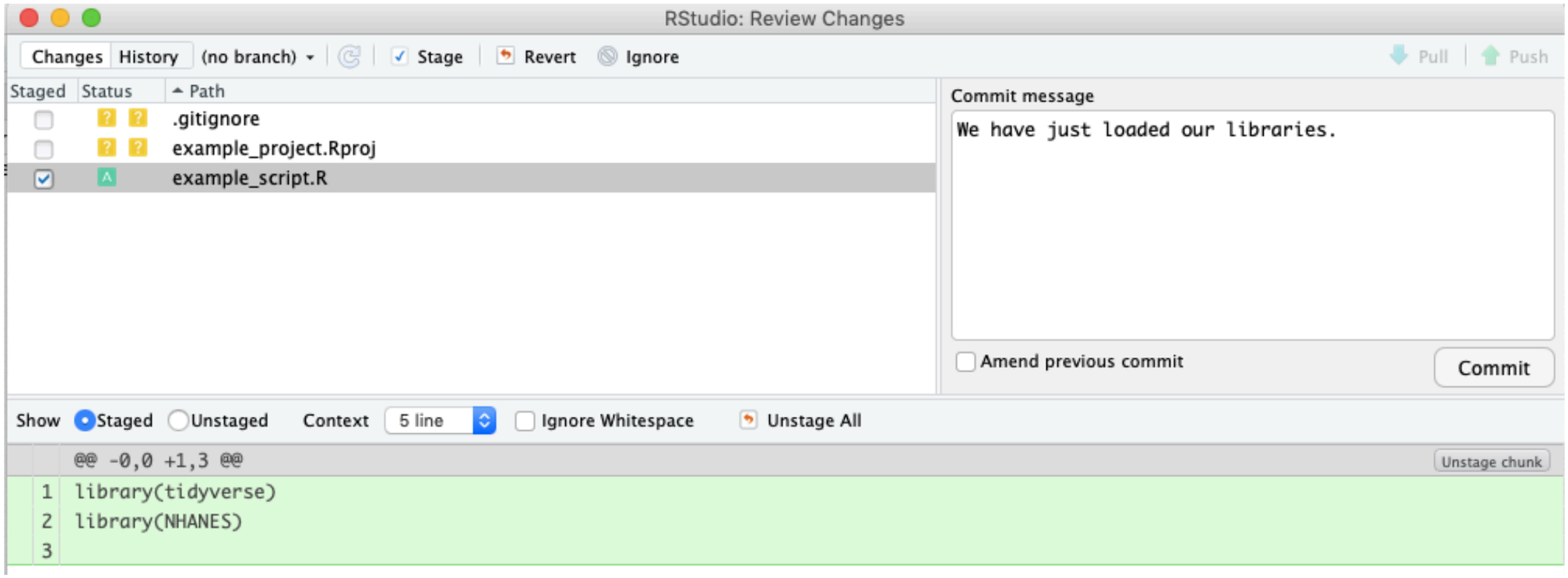
- Git works by keeping track of changes you make to your script during each save - you can mark a point at which you want git to remember your script by ‘committing’ the script at the time point you save it…




- In this example, I’ve written two lines of code then saved my script - I then click “Commit” under Version Control in the Tools menu which then produces the following…



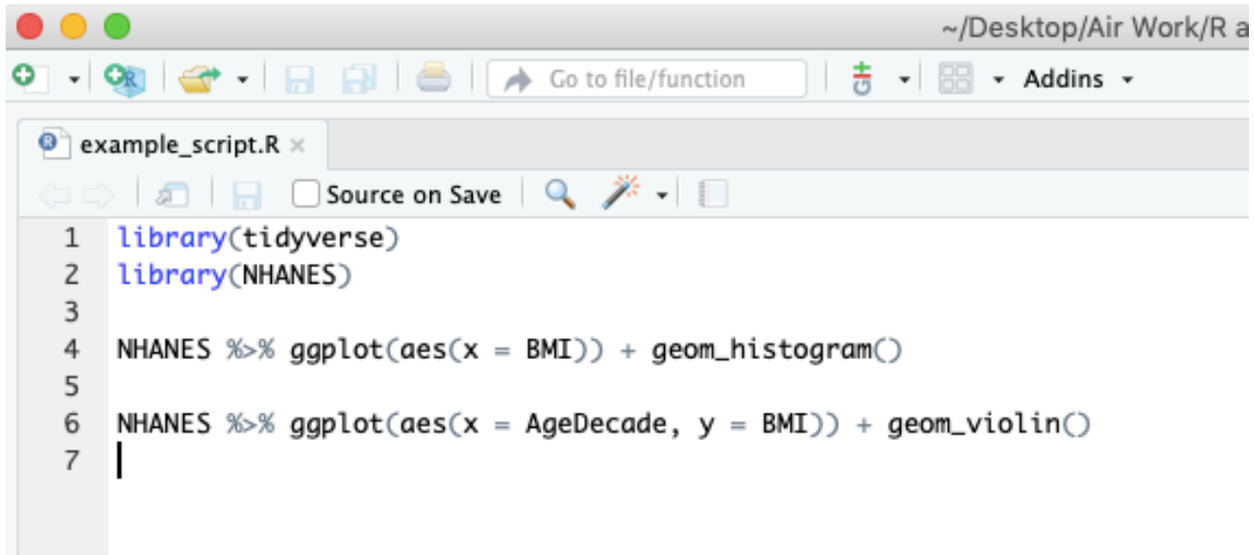

- We then need to put a tick on the box of what we want to commit and then write some sort of helpful message capturing the point in writing the script that we're making the commit.



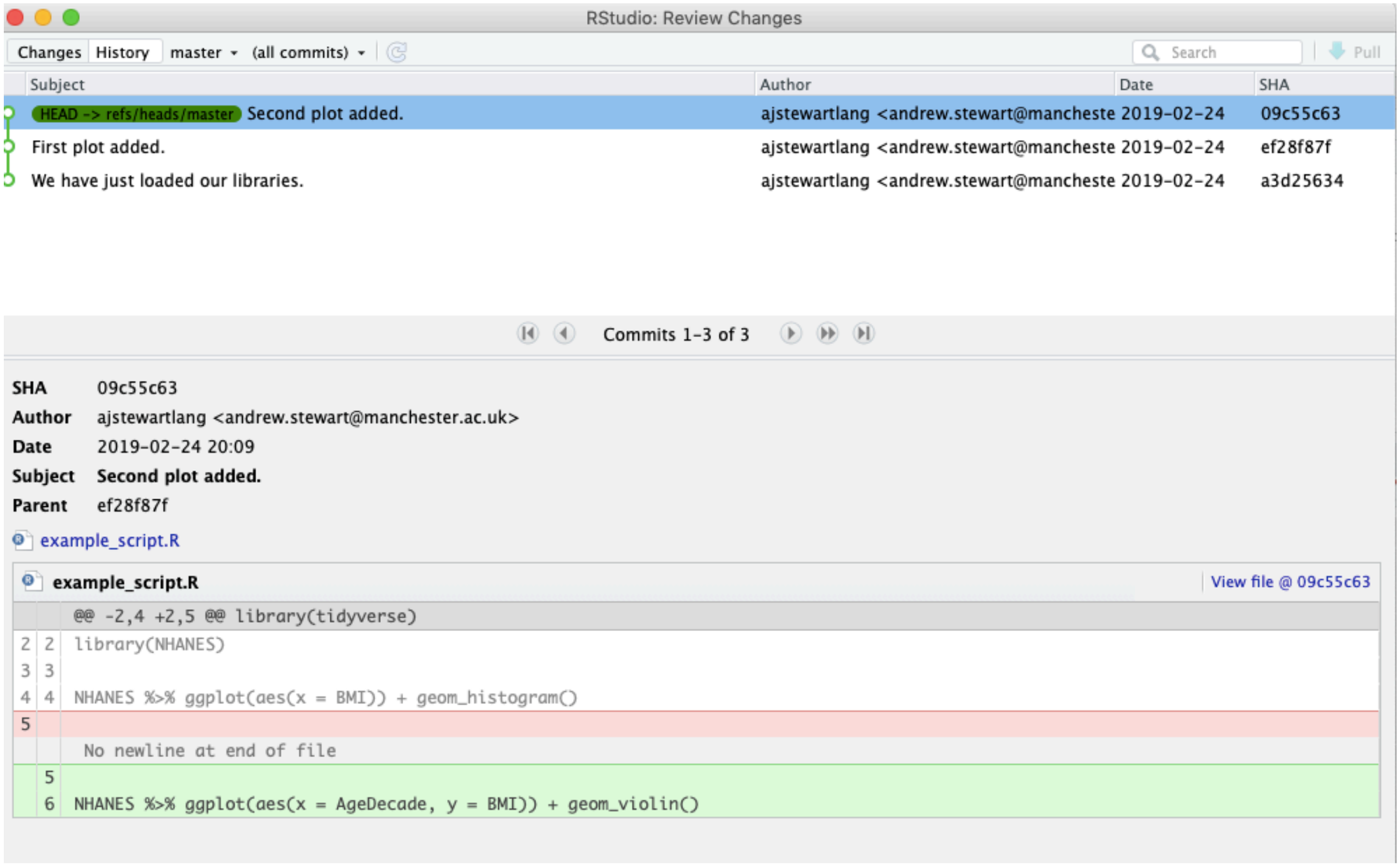

- We have put the tick mark in the box associated with the file we want Version Control on - and I've added a line in the Commit message box to indicate at what stage of writing the script I have made the commit.  Click the Commit button once you're ready to capture this.

- We get a summary of our Commit.  OK, I'm going to add two more lines of code, save after each then, and will make a commit after each save.  The first line of code will generate one plot, and the second line a different plot.

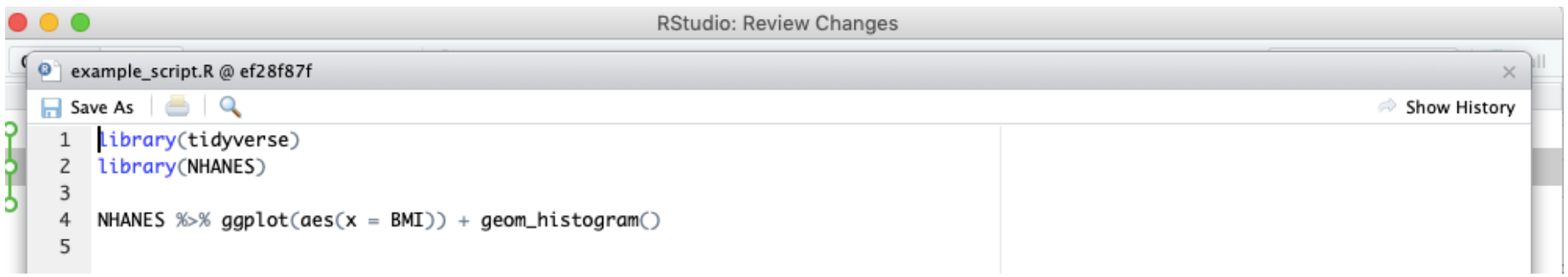

- My 'final' script looks this:



- But I can revert (i.e., go back) to a previous version of the script - go to the Version Control option in the Tool menu and click on “History”. I can see each of my three commits…



- If I click on the second commit, and then click on the View file link, I can see the version of the script associated with that commit:




- This commit was made just after I added the first plot, and before I added the second one.  If I wanted to, I could just copy this file into my script window, and I'm back to where I was before I added that second plot!

- Alternatively, if I had saved my file but not made a commit, I could simply select "Revert" in the Tools - Version Control menu to go back to the file in its previous state.

- Remember you can only make a commit **after** you've saved your file. Git works by tracking how your saved file changes over time.

- Git is *hugely* powerful.  We've only looked at how you can use it to keep track of changes you are making to your own scripts. If this is all you use it for, you will still find it hugely helpful!

- Git comes into its own when you're working on collaborative projects - you can use git to 'fork' (i.e., make a copy of) someone else's code.  You can modify their code - maybe by adding a new section and then you can ask them to 'pull' your fork back into their 'master' - thus allowing your code to be added to what they're working on.

- If you want to get serious with git, create an account on GitHub and then 'fork' someone else's (maybe a friend's) repository (aka repo).  Modify the code and then send a 'pull' request for your changes to be incorporated into their code.

Lots more information here 👇

`https://happygitwithr.com`

# This Afternoon

- This afternoon the worksheet contains a number of possible exercises to work on - you probably won't have time to do them all so start with the one you think will be more interesting/useful to you!