5. Permutation Tests (i.e. replication)

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Auckland, November 2013

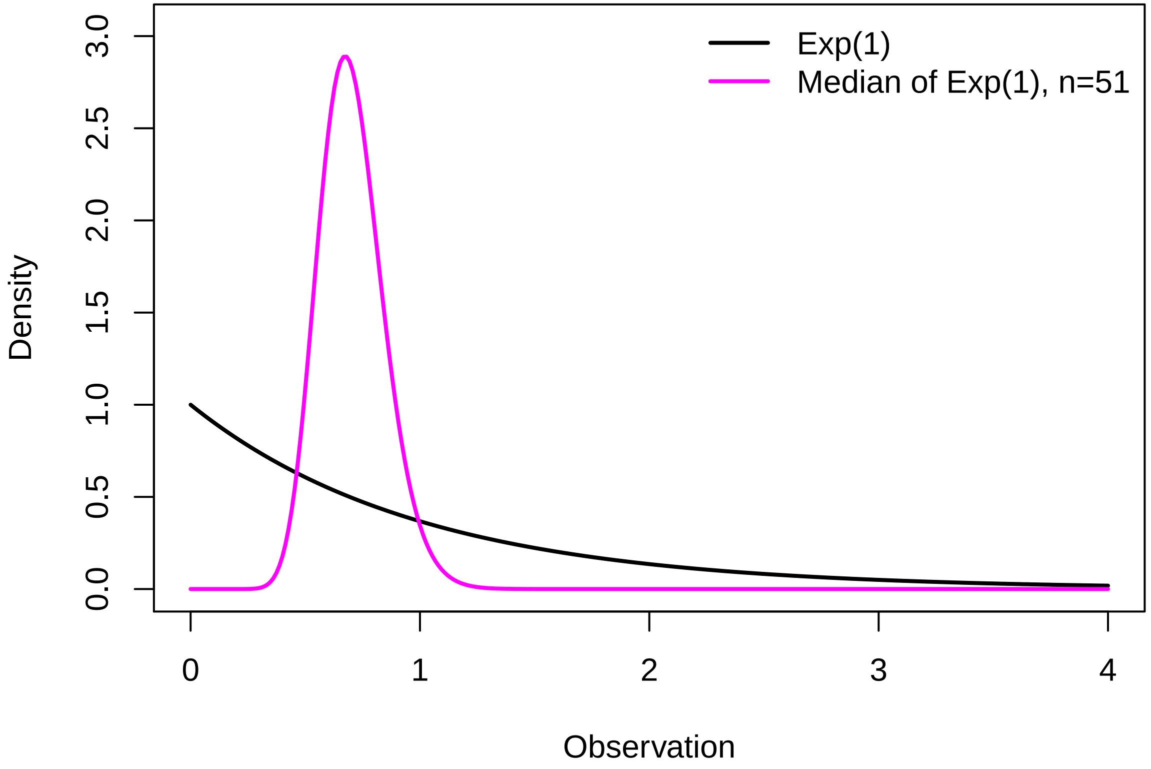
Overview

- Simulation
- Permutation tests
- Power (briefly)
- Smallest p-value across multiple models
- Cautionary notes, which we won't get to

Simulation: easier than doing maths

A question from analysis of survival traits – and its answer!

What is the expected value of the median of a sample, size $n = 51$*, of independent data from* $Exp(1)$*?*
What is its variance?



Simulation: easier than doing maths

The picture didn't make it obvious? Here are the *exact* answers;

$$
\begin{aligned}
\mathbb{E}[\,\text{Median}_{51}\,] &= \frac{2178178936539108674153}{3099044504245996706400} \\
\mathbb{E}[\,\text{Median}^2_{51}\,] &= \frac{246728231606366796745923323213925797680195 9}{480203841964865774900127881537982390048000 0}
\end{aligned}
$$

These are 0.70286 and 0.51380 to 5 d.p. – so the variance is $0.51380 - 0.70286^2 = 0.01978$.

- Yes, there are 'pretty' answers here
- In general there aren't – but the 'expectation' ($\mathbb{E}[\ldots]$) terms just mean averaging over lots of datasets – which is easy, with a computer
- We can get a good-enough answer very quickly

Simulation: easier than doing maths

We'll write code that;

1. Generates samples of size $n = 51$ from $Exp(1)$
2. Calculates their median and returns this number
3. Replicate steps 1 and 2 many times, then work out the mean and variance of the stored numbers

Steps 1 and 2 are inside the curly brackets;

```
set.seed(4)
many.medians <- replicate(10000, {
   mysample <- rexp(n=51, rate=1)   # take a sample, size 51
   median(mysample)                 # calcuate & output its median
   })
```

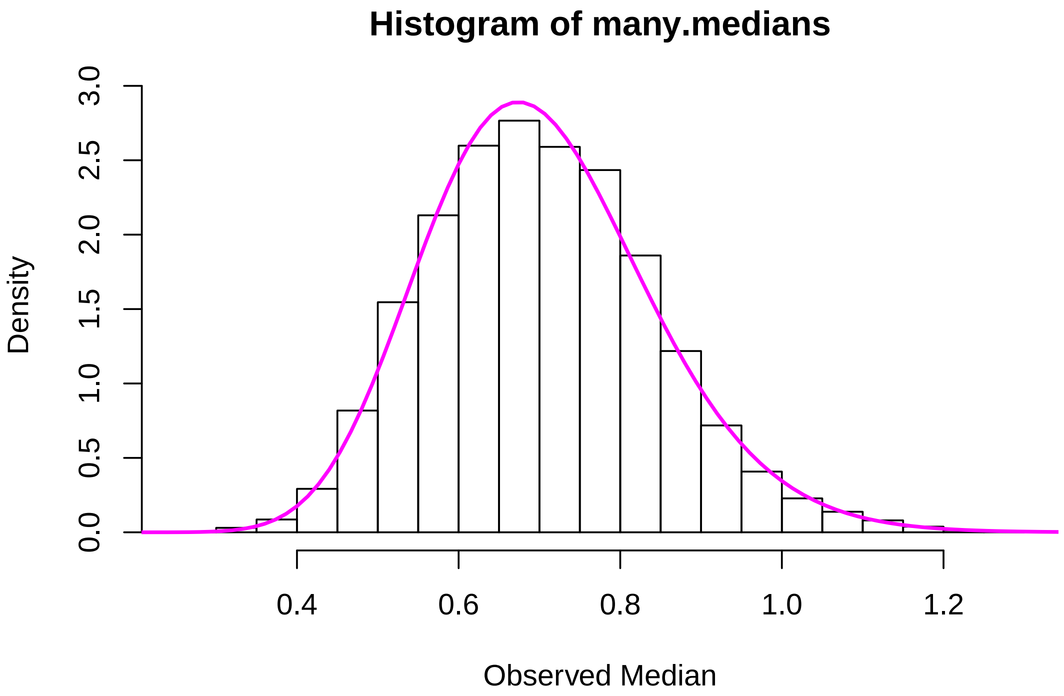
The function `set.seed()` tells R where to start its random-number generator – this is important, as it means we can repeat the code and get the same answers. Choose any 'seed' you like.

Simulation: easier than doing maths

To finish, take mean & variance of our 10000 sample medians;

```
> mean(many.medians)
[1] 0.702171          # exact answer is 0.70286
> var(many.medians)
[1] 0.01955728        # exact answer is 0.01978
```

NB: for large-enough values of 10000, we could work basically *anything* about the sample median, with little extra work;



Simulation: easier than doing maths

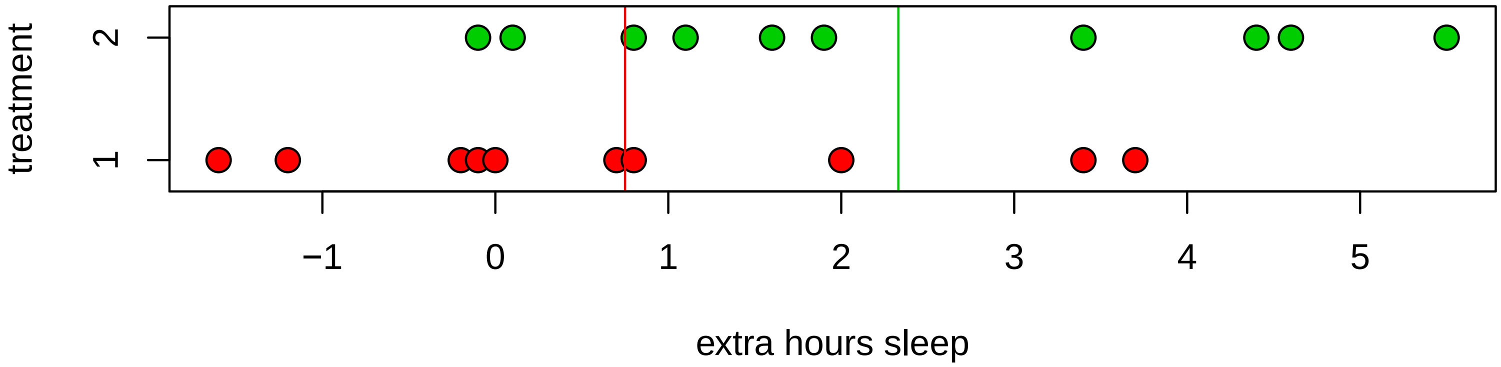
Notes on the coding;

- Yes, you could write a `for()` loop. (See Session 6, or `?Control`) But this approach helps break down the job into manageable pieces – then *finally* deal with the looping
- This approach is also easier to setup – just create `many.medians` and easier to edit afterwards, e.g. changing the value of 10000. Using `for()` loops, it's surprisingly easy to mess this up (more in Session 6)
- By default, the last object evaluated in a function is what it returns. Can also use `return()`
- We used `rexp()`, see also `rnorm()`, `rgamma()`, `rbinom`, `rpois()` etc etc

Example: permutation test

A classical statistical question: are the data we've observed *unexpected*, if there's nothing going on?

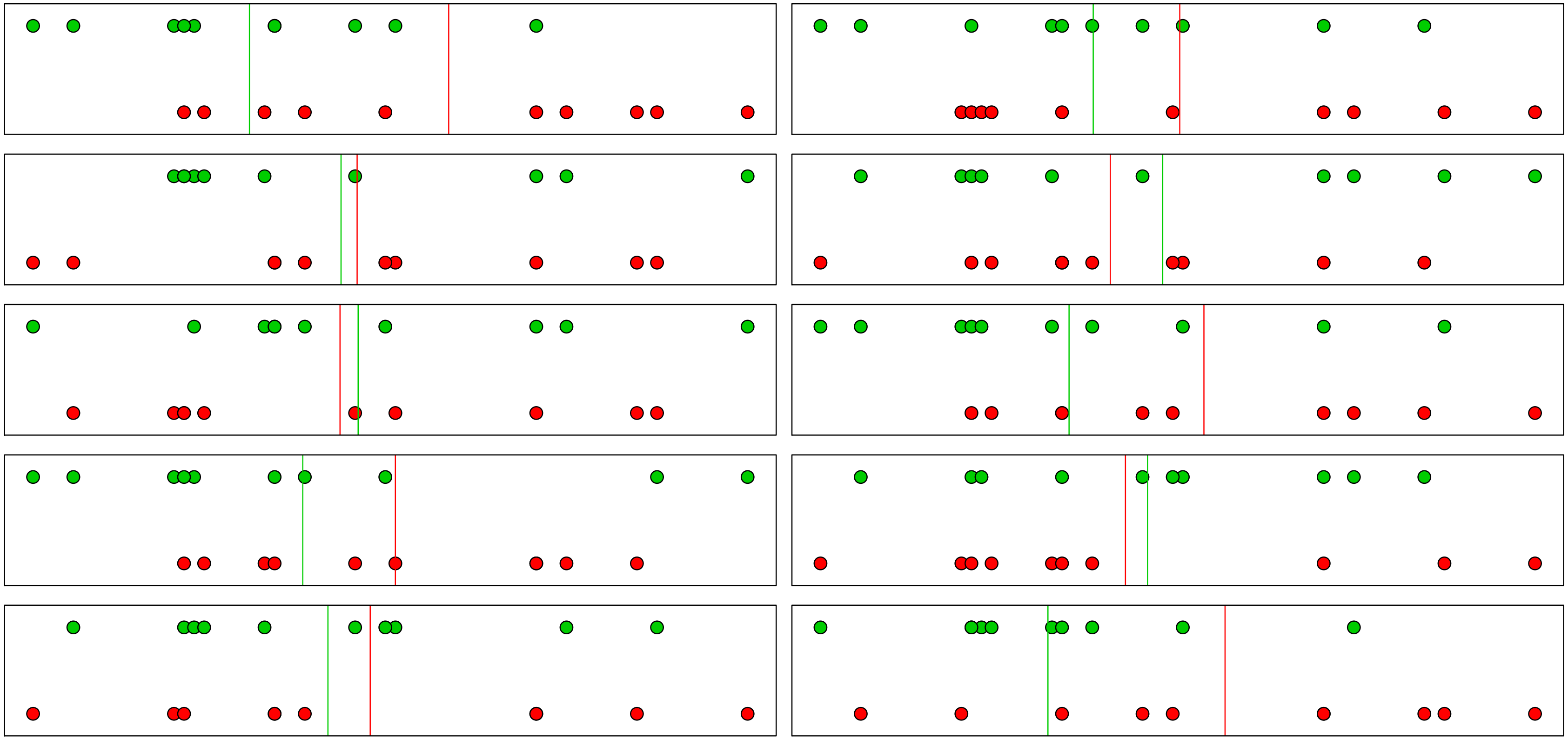
An example where we can answer this is the `sleep` data, from a small clinical trial;



- 10 subjects per group
- Groups receive different treatments, we record how many hours sleep they get, compared to baseline
- Mean extra hours sleep is higher in group 2 (2.33 hrs vs 0.75 hrs, so difference is 1.58 hrs)

Example: permutation test

What if there were nothing going on*, i.e. what if any differences in mean were just chance? If so, the data we saw would be just as likely as that obtained assigning the group labels *at random*;



* Formally, what if the *null hypothesis* of equal means held, in the population from which this data has been sampled?

Example: permutation test

To measure how unexpected our data is, we compute the red/green difference in means for many of these *permutations*, and see how the *observed* data compares.

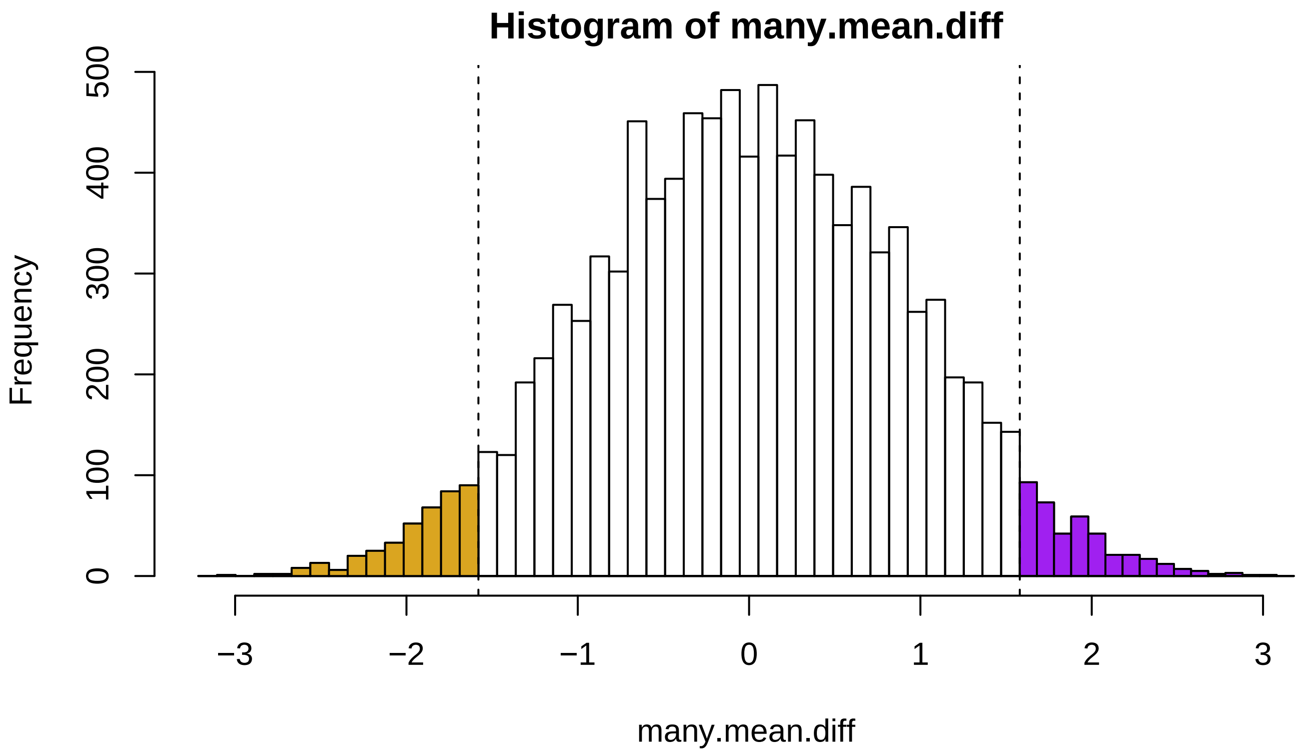
```
orig.mean.diff <-
   mean(sleep$extra[group==2])        - mean(sleep$extra[group==1])
orig.mean.diff

set.seed(4)
many.mean.diff <- replicate(10000,{
  group.shuffle <- sample(sleep$group)
  mean(sleep$extra[group.shuffle==2]) - mean(sleep$extra[group.shuffle==1])
  })
```

- `sample()` returns a random shuffle of a vector
- The same calculation is made, for the original data and the shuffled versions; the difference in means is called the *test statistic*

Example: permutation test

How does original data (w/ mean diff=1.58) compare to these?



```
> table(many.mean.diff>orig.mean.diff)
FALSE  TRUE
 9601   399
> mean(many.mean.diff>orig.mean.diff)
[1] 0.0399
> mean(abs(many.mean.diff)>abs(orig.mean.diff))
[1] 0.0789
```

Example: permutation test

- The proportion of sample in the RH tail is a (valid) p-value for a one-tailed test, where the alternative is that green $>$ red. $p = 0.04$, here
- The proportion in both tails is the p-value for a two-tail test; $p = 0.079$
- There is some 'Monte Carlo' error in these p-values; roughly ±0.004 here, i.e. 2 decimal places in p. If that's not good enough, use more permutations. (Here, could use all 184,756 – but in larger samples it's not possible)

For a quicker but *somewhat* approximate version of this test;

```
> t.test(extra˜group, data=sleep)
Welch Two Sample t-test
data:  extra by group
t = -1.8608, df = 17.776, p-value = 0.07939
alternative hypothesis: true difference in means is not equal to 0
```

The t-test makes fewer assumptions than many people think!

How many permutations?

With 10000 permutations the smallest possible p-value is 0.0001, and the uncertainty near $p = 0.05$ is about $\pm 0.4\%$

If we have multiple testing we may need much more precision.

Using 100,000 permutations reduces the uncertainty near $p = 0.05$ to $\pm 0.1\%$ and allows accurate p-values as small as 0.00001.

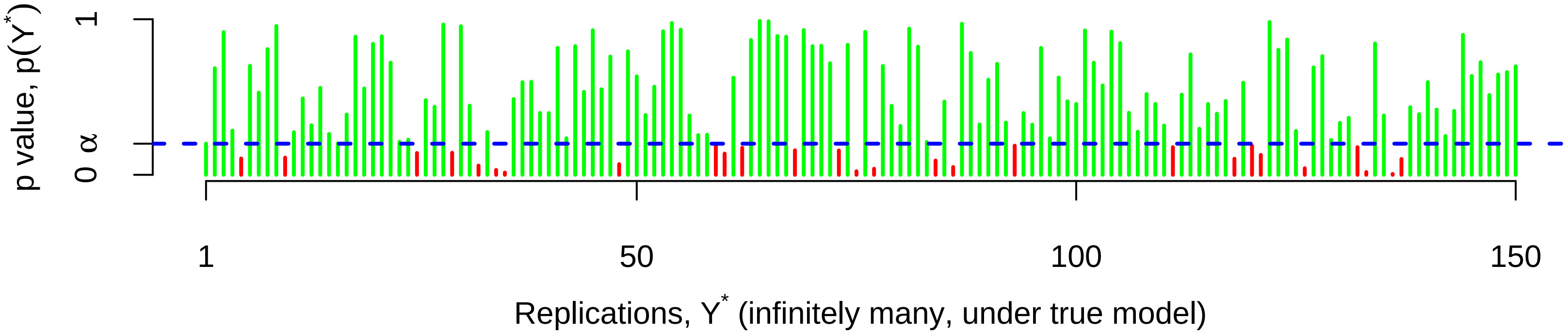
A useful strategy is to start with 1000 permutations and continue to larger numbers only if p is small enough to be interesting, eg $p < 0.1$.

Parallel computing of permutations is easy: just run R on multiple computers.

Example: power calculation

A reminder: statistical tests **reject** H_0 whenever $p(Y) < \alpha$; α is known as the **size** or **level** of the test; it is the *probability* of declaring a signal when none is present. (By convention, we almost always use $\alpha = 0.05$.)

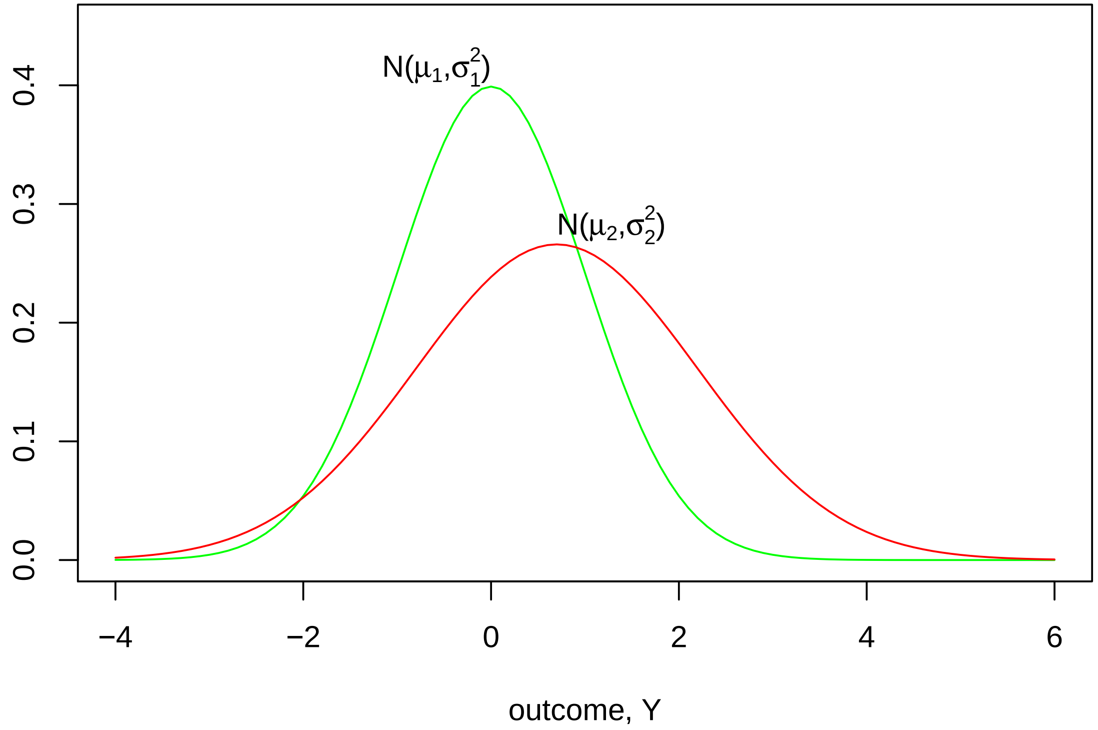
The **power** of the test is the *probability* you reject H_0, (i.e. get $p(Y) < \alpha$) when a signal is truly present.



Note: *probabilities* are means of binary (0/1) variables.

Example: power calculation

To evaluate power, for a comparison of two groups – like the sleep study;



- Assume outcomes are Normal, means μ_1, μ_2, SDs σ_1, σ_2
- Sample sizes n_1, n_2 in each group
- Use unequal-variance two-sided t-test for analysis

Example: power calculation

Using `replicate()` to do the work;

```
do.one <- function(n1, n2, mu1, mu2, s1, s2, alpha){
   y1 <- rnorm(n1, mu1, s1)
   y2 <- rnorm(n2, mu2, s2)
   t.test(y1, y2)$p.value < alpha # default is unequal variance
}

set.seed(4)
bigB <- 10000
mean(replicate(bigB, do.one(20, 20, 0, 0.7, 1, 1.5, 0.05)))
```

- Mean here is 3895/10000 = 0.3895, i.e. about 40% power. Precision to multiple decimal places matters much less than earlier example, calculating p
- For equal variances, can use `power.t.test()`, i.e. built-in maths-only version
- This version is slower to compute – but much more flexible, e.g. regression-based analyses, multi-step analyses, any distribution/design you like

Debugging

R error messages are sometimes hard to follow, when the error occurs in a low-level function. To see what happened after an error, `traceback()` reports the entire call stack, which is useful for seeing where things went wrong.

For example, in the permutation test example, suppose our outcome variable were actually a data frame with one column rather than a vector:

```
> wrong.extra <- as.data.frame(sleep$extra) # easy mistake!
> many.mean.diff <- replicate(10000,{
+   group.shuffle <- sample(sleep$group)
+   mean(wrong.extra[group.shuffle==2]) - mean(wrong.extra[group.shuffle==1])
+   })
Error in ‘[.data.frame‘(wrong.extra, group.shuffle == 2) :
  undefined columns selected
```

We didn’t know we were calling `‘[.data.frame‘`, so the message appears opaque and unhelpful.

Debugging

```
> traceback()
8: stop("undefined columns selected")
7: ‘[.data.frame‘(wrong.extra, group.shuffle == 2)
6: wrong.extra[group.shuffle == 2]
5: mean(wrong.extra[group.shuffle == 2]) at #3
4: FUN(c(0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L,
3: lapply(X = X, FUN = FUN, ...)
2: sapply(integer(n), eval.parent(substitute(function(...) expr)),
       simplify = simplify)
1: replicate(10000, {
       group.shuffle <- sample(sleep$group)
       mean(wrong.extra[group.shuffle == 2]) - mean(wrong.extra[group.shuffle == 1
   })
```

This mean the problem happens in our code `mean(wrong.extra[group.shuffle == 2])` and is a problem with computing `wrong.extra[group.shuffle == 2]`.

We want to have a look at `wrong.extra` and `group.shuffle`...

Debugging

The post-mortem debugger lets you look inside the code where the error occurred;

```
> options(error=recover)
> many.mean.diff <- replicate(10000,{
+   group.shuffle <- sample(sleep$group)
+   mean(wrong.extra[group.shuffle==2]) - mean(wrong.extra[group.shuffle==1])
+   })
Error in ‘[.data.frame‘(wrong.extra, group.shuffle == 2) :
  undefined columns selected

Enter a frame number, or 0 to exit

1: replicate(10000, {
    group.shuffle <- sample(sleep$group)
    mean(wrong.ex
2: sapply(integer(n), eval.parent(substitute(function(...) expr)), simplify =
3: lapply(X = X, FUN = FUN, ...)
4: FUN(c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
5: #3: mean(wrong.extra[group.shuffle == 2])
6: wrong.extra[group.shuffle == 2]
7: ‘[.data.frame‘(wrong.extra, group.shuffle == 2)
```

We want 6, the last step of ‘our’ code

Debugging

```
Called from: eval(substitute(browser(skipCalls = skip), list(skip = 7 - which)),
    envir = sys.frame(which))
Browse[1]> ls()
character(0)
Browse[1]> str(wrong.extra)
'data.frame':   20 obs. of  1 variable:
 $ sleep$extra: num  0.7 -1.6 -0.2 -1.2 -0.1 3.4 3.7 0.8 0 2 ...
Browse[1]> is.vector(wrong.extra)
[1] FALSE
Browse[1]> c
```

As well as data in unexpected formats, watch out for ‘weird’ data that might lead to e.g. missing values in regression output.

Finally; turn the post-mortem debugger off with

```
options(error=NULL) # turn it off! turn it off!!!
```

Minimum p-value

Little point in permutation test for the mean: same result as t-test

Permutation test is useful when we do not know how to compute the distribution of a test statistic.

Suppose we test additive effects of 8 SNPs, one at a time, and we want to know if the most significant association is real.

For any one SNP the z-statistic from a logistic regression model has a Normal distribution.

We need to know the distribution of the most extreme of eight z-statistics. This is not a standard distribution, but a permutation test is still straightforward.

Minimum p-value

```
dat <- data.frame(y=rep(0:1,each=100),
SNP1=rbinom(200,size=2,prob=.1), SNP2=rbinom(200,size=2,prob=.2),
SNP3=rbinom(200,size=2,prob=.2), SNP4=rbinom(200,size=2,prob=.4),
SNP5=rbinom(200,size=2,prob=.1), SNP6=rbinom(200,size=2,prob=.2),
SNP7=rbinom(200,size=2, prob=.2), SNP8=rbinom(200,size=2,prob=.4))

> head(dat)
  y SNP1 SNP2 SNP3 SNP4 SNP5 SNP6 SNP7 SNP8
1 0    0    0    0    0    0    1    0    0
2 0    0    1    0    1    0    1    0    2
3 0    0    1    0    1    1    0    0    0
4 0    0    0    1    1    0    0    0    0
5 0    0    1    0    1    1    0    0    0
6 0    0    0    0    1    0    1    0    1
```

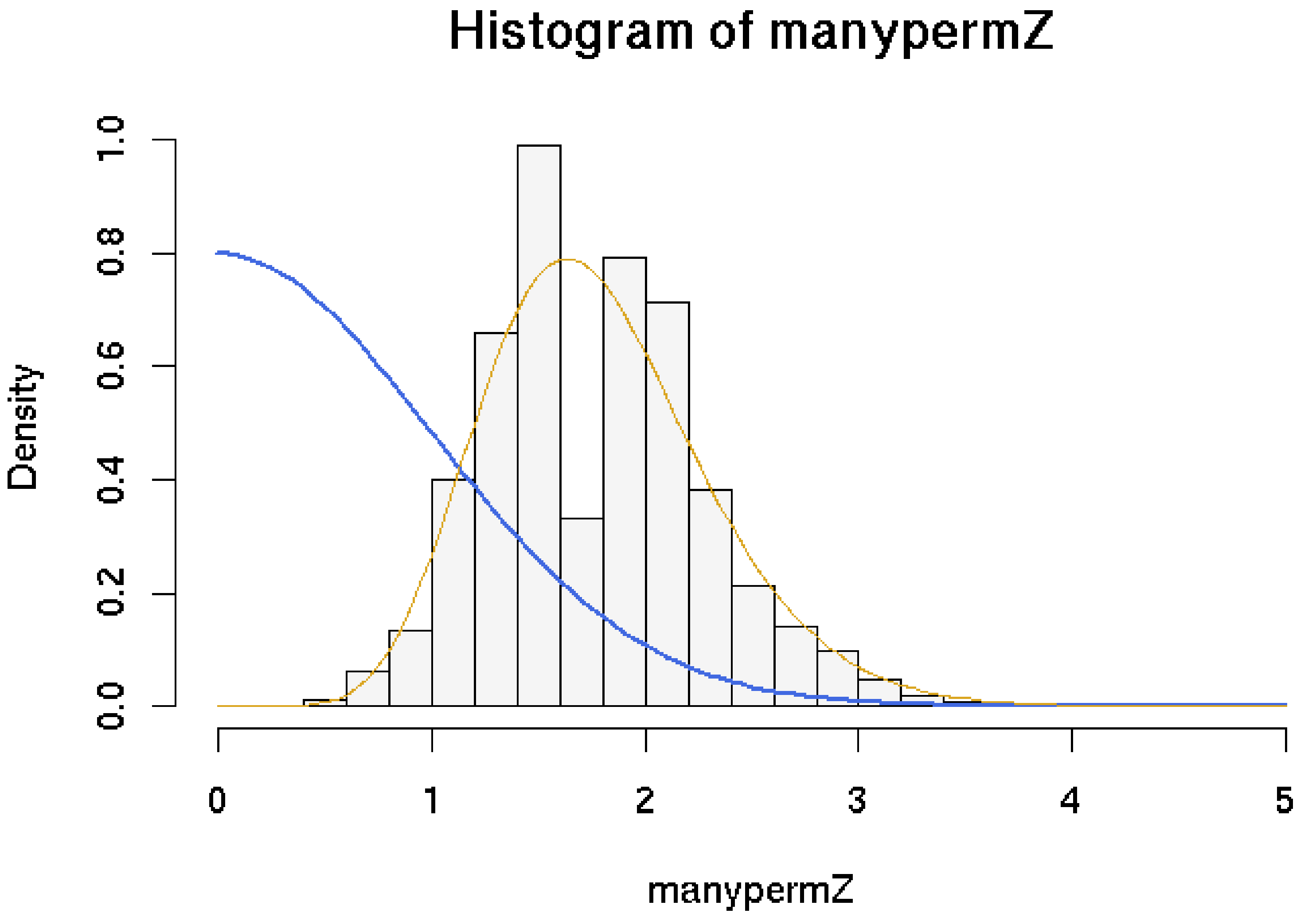
Minimum p-value

```
oneZ<-function(outcome, snp){
    model <- glm(outcome~snp, family=binomial)
    coef(summary(model))["snp","z value"]
}
maxZ<-function(outcome, snps){
    allZs <- sapply(snps,
         function(this.snp){ oneZ(outcome, snp=this.snp) })
    max(abs(allZs))
}
```

```
true.maxZ<-maxZ(outcome=dat$y, snps=dat[,-1])
```

```
manypermZ<-replicate(10000,
       maxZ(outcome=sample(dat$y), snps=dat[,-1]))
```

Minimum p-value



Minimum p-value

The histogram shows the permutation distribution for the maximum Z-statistic.

The blue curve is the theoretical distribution for one Z-statistic

The yellow curve is the theoretical distribution for the maximum of eight independent Z-statistics.

Clearly the multiple testing is important: a Z of 2.5 gives $p =$ 0.012 for a single test but $p =$ 0.075 for the permutation test.

The theoretical distribution for the maximum has the right range but the permutation distribution is quite discrete. The discreteness is more serious with small sample size and rare SNPs.

[The theoretical distribution is not easy to compute except when the tests are independent.]

More debugging

Permutation tests on other people's code might reveal a lack of robustness.

For example, a permutation might result in all controls being homozygous for one of the SNPs and this might give an error message

We can work around this with `tryCatch()`

```
oneZ<-function(outcome, snp){
    tryCatch({model <- glm(outcome~snp, family=binomial())
              coef(summary(model))["snp","z value"]},
       error=function(e) NA
       )
}
```

Now `oneZ()` will return `NA` if there is an error in the model fitting.

Caution: wrong null

Permutation tests cannot solve all problems: they are valid only when the null hypothesis is ‘no association’

Suppose we are studying a set of SNPs that each have some effect on outcome and we want to test for interactions (epistasis).

Permuting the genotype data would break the links between genotype and outcome and created shuffled data with no main effects of SNPs.

Even if there are no interactions the shuffled data will look different from the real data.

Caution: weak null hypothesis

A polymorphism could increase the variability of an outcome but not change the mean.

In this case the strong null hypothesis is false, but the hypothesis of equal means is still true.

- If we want to detect this difference the permutation test is unsuitable because it has low power
- If we do not want to detect this difference the permutation test is invalid, because it does not have the correct Type I error rate.

When groups are the same size the Type I error rate is typically close to the nominal level, otherwise it can be too high or too low.

To illustrate this we need many replications of a permutation test. We will do 1000 permutation tests for a mean, each with 1000 permutations.

```
meandiff<-function(x,trt){
mean(x[trt==1])-mean(x[trt==2])
}
meanpermtest<-function(x,trt,n=1000){
observed<-meandiff(x,trt)
perms<-replicate(n, meandiff(x, sample(trt)))
mean(abs(observed)>abs(perms))
}

trt1<-rep(c(1,2),c(10,90))

perm.p<-replicate(1000, {
          x1<-rnorm(100, 0, s=trt1)
          meanpermtest(x1,trt1)})

table(cut(perm.p,c(0,.05,.1,.5,.9,.95,1)))/1000
```

```
  (0,0.05] (0.05,0.1]  (0.1,0.5]  (0.5,0.9] (0.9,0.95]   (0.95,1]
        86         99        564        244          6          0
```

The p-values are too small, relative to a uniform distribution. If we reverse the standard errors we get

```
  (0,0.05] (0.05,0.1]  (0.1,0.5]  (0.5,0.9] (0.9,0.95]   (0.95,1]
        27         28        275        354         67        249
```

If the two groups each have 50 observations we get

```
  (0,0.05] (0.05,0.1]  (0.1,0.5]  (0.5,0.9] (0.9,0.95]   (0.95,1]
        50         45        403        407         52         43
```

which is much better.