Simulating Power

Jeffrey Hughes

2017-10-23

Simulating Power with the paramtest Package

For simple statistical models (e.g., *t*-test, correlation), calculating the estimated power can be done analytically (for example, one can use the 'pwr' package). But for more complex models, it is difficult to provide a good estimate of power without the use of simulation. Simulations repeatedly generate random data based on one's predefined model, then analyze each data set and count the proportion of results that are significant. That proportion is the estimated power for the model.

The 'paramtest' package makes this process simple by providing a general-purpose, flexible function to perform simulations. You simply create a function that generates the data and analyses you want, and then use one of the functions, such as `grid_search()`, to run your user-defined function repeatedly and collate the results. The `grid_search()` function allows you to easily run your function iteratively while varying particular parameters, so you can test the sensitivity of your analyses or how these parameters change the results.

```
# load all packages used in this vignette
library('paramtest')
library('pwr')
library('ggplot2')
library('knitr')
library('nlme')
library('lavaan')
library('dplyr')
```

Example: Simulating a *t*-test with paramtest

If we wanted to estimate the power for a two-sample *t*-test, we could calculate it analytically using the 'pwr' package:

```
pwr.t.test(n=50, d=.5, type='two.sample')
#>
#>      Two-sample t test power calculation
#>
#>               n = 50
#>               d = 0.5
#>       sig.level = 0.05
#>           power = 0.6968934
#>     alternative = two.sided
#>
#> NOTE: n is number in *each* group
```

We can see that the estimated power when Cohen's *d* = .50 and *n* = 50 per cell is approximately .70. Simulating power in this simple case is likely overkill, but this example will demonstrate that simulations provide comparable results, at least for this model.

```
# create user-defined function to generate and analyze data
t_func <- function(simNum, N, d) {
    x1 <- rnorm(N, 0, 1)
    x2 <- rnorm(N, d, 1)

    t <- t.test(x1, x2, var.equal=TRUE)  # run t-test on generated data
    stat <- t$statistic
    p <- t$p.value

    return(c(t=stat, p=p, sig=(p < .05)))
        # return a named vector with the results we want to keep
}

power_ttest <- run_test(t_func, n.iter=5000, output='data.frame', N=50, d=.5)  # simulate data
results(power_ttest) %>%
    summarise(power=mean(sig))
```

```
#>    power
#> 1 0.7024
```

We first create a function that simulates normally distributed data for two groups, and performs a *t*-test. The *t* statistic and *p*-value are then returned as a named vector, along with a boolean value determining whether the test is significant or not. Finally, we use the `run_test()` function to perform 5000 simulations, and then summarize the 'sig' value, which (by default) calculates the mean, giving us the proportion of simulations that were significant. This number agrees very closely with the analytic solution above.

Varying parameters

While in the above example we used `run_test()` because we were not varying any parameters, with the 'paramtest' package we can also use a function called `grid_search()` to easily vary parameters. For example, using the same *t*-test simulation as above:

```
# give 'params' a list of parameters we want to vary;
# testing at N=25, N=50, and N=100
power_ttest_vary <- grid_search(t_func, params=list(N=c(25, 50, 100)),
    n.iter=5000, output='data.frame', d=.5)
results(power_ttest_vary) %>%
    group_by(N.test) %>%
    summarise(power=mean(sig))
```

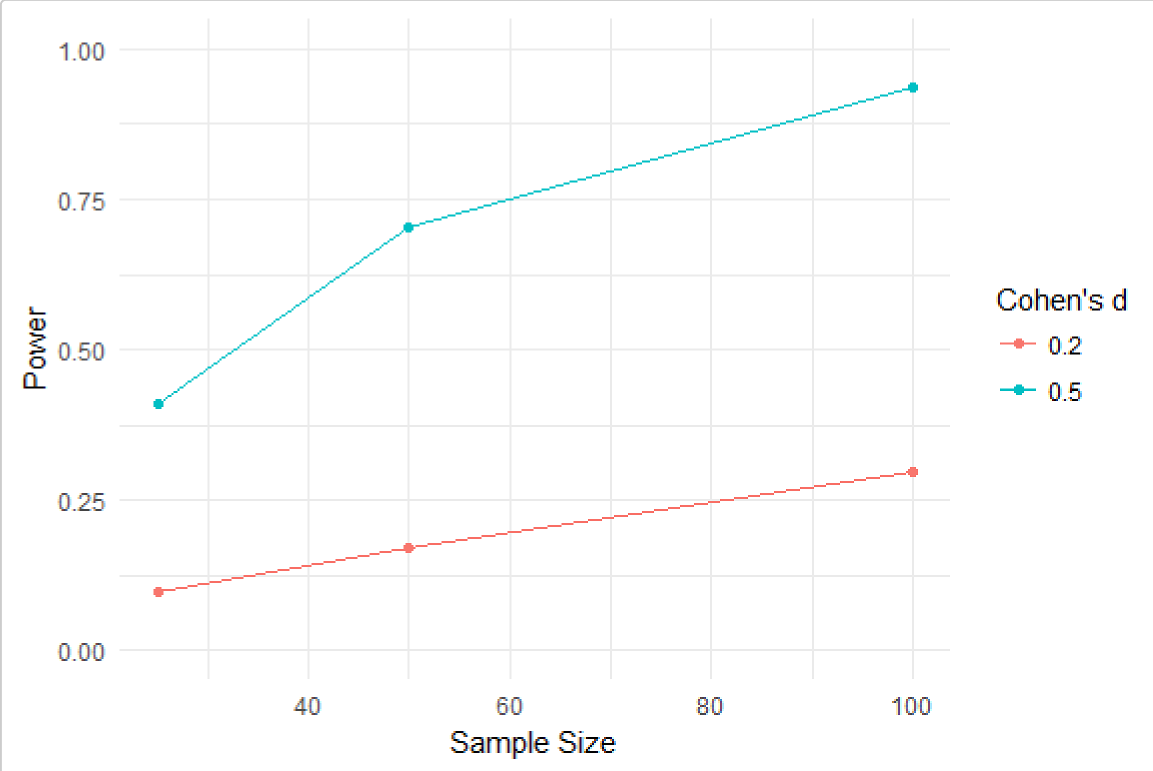
N.test	power
25	0.402
50	0.703

N.test	power
100	0.940

We were easily able to run simuations for three different sample sizes: 25 per cell, 50 per cell, and 100 per cell. The `summary()` function shows us the proportion of simulations that were significant for each sample size. Clearly, power increases as *N* increases. If we wanted to vary across two separate parameters, that is easy to do as well:

```
# varying N and Cohen's d
power_ttest_vary2 <- grid_search(t_func, params=list(N=c(25, 50, 100), d=c(.2, .5)),
    n.iter=5000, output='data.frame')
power <- results(power_ttest_vary2) %>%
    group_by(N.test, d.test) %>%
    summarise(power=mean(sig))
print(power)
ggplot(power, aes(x=N.test, y=power, group=factor(d.test), colour=factor(d.test))) +
    geom_point() +
    geom_line() +
    ylim(c(0, 1)) +
    labs(x='Sample Size', y='Power', colour="Cohen's d") +
    theme_minimal()
```

N.test	d.test	power
25	0.2	0.098
25	0.5	0.409
50	0.2	0.171
50	0.5	0.704
100	0.2	0.298
100	0.5	0.936



Note that `grid_search()` will fully cross all parameters: the three sample sizes are tested at *d* = .2, and at *d* = .5. So be careful when adding new parameters, as this can greatly increase the total number of simulations to be run!

Parallel processing

In order to cut down on the time taken to run simulations, the ‘paramtest’ package supports parallel processing via the ‘parallel’ package. If your system has multiple processor cores or supports multithreading, you can incorporate this into your function call:

```
# time with no parallel processing (your mileage may very greatly)
system.time(power_ttest_vary3 <- grid_search(t_func,
    params=list(N=c(25, 50, 100), d=c(.2, .5)), n.iter=5000, output='data.frame'))
```

```
#>    user  system elapsed 
#>    4.33    0.03    4.36
```

```
# using parallel processing; I am using a Windows system, so I use parallel='snow';
# see the documentation for the 'parallel' package for more details
system.time(power_ttest_vary4 <- grid_search(t_func,
    params=list(N=c(25, 50, 100), d=c(.2, .5)), n.iter=5000, output='data.frame', parallel='snow',
ncpus=4))
```

```
#>    user  system elapsed 
#>    0.42    0.03    2.25
```

Theoretically, when using *n* cores/threads, your code should take 1/*n* the amount of time. In practice, there is some overhead to setting up the parallel threads, so you may not get quite such an improvement. But the proportion of time saved should grow the longer the overall job is.

Bootstrapping

In addition to running a function that creates simulated data, the 'paramtest' package can also handle bootstrapping, where you have data that you want to randomly sample from. This features relies on the 'boot' package:

```
# user function must take data and indices as first two arguments; see 'boot'
# package documentation for more details
t_func_boot <- function(data, indices) {
    sample_data <- data[indices, ]
    treatGroup <- sample_data[sample_data$group == 'trt2', 'weight']
    ctrlGroup <- sample_data[sample_data$group == 'ctrl', 'weight']

    t <- t.test(treatGroup, ctrlGroup, var.equal=TRUE)
    stat <- t$statistic
    p <- t$p.value

    return(c(t=stat, p=p, sig=(p < .05)))
}

# example using built-in dataset PlantGrowth
power_ttest_boot <- run_test(t_func_boot, n.iter=5000, output='data.frame', boot=TRUE,
    bootParams=list(data=PlantGrowth))
results(power_ttest_boot) %>%
    summarise(power=mean(sig))
```

```
#>    power
#> 1 0.5376
```

Sample code for various statistical models

Linear models

We can use `grid_search()` to calculate the power for a particular coefficient in a linear model:

```
lm_test <- function(simNum, N, b1, b0=0, xm=0, xsd=1) {
    x <- rnorm(N, xm, xsd)
    y <- rnorm(N, b0 + b1*x, sqrt(1 - b1^2))  # var. approx. 1 after accounting
                                               # for explained variance by x
    model <- lm(y ~ x)

    # pull output from model
    est <- coef(summary(model))['x', 'Estimate']
    se <- coef(summary(model))['x', 'Std. Error']
    p <- coef(summary(model))['x', 'Pr(>|t|)']
```

```
    return(c(xm=mean(x), xsd=sd(x), ym=mean(y), ysd=sd(y), est=est, se=se, p=p,
        sig=(p < .05)))
}

# we vary N at 200 and 300; we are also setting coefficient of x predicting
# y to be approx. .15 across all simulations
power_lm <- grid_search(lm_test, params=list(N=c(200, 300)), n.iter=5000, output='data.frame',
b1=.15,
    parallel='snow', ncpus=4)
results(power_lm) %>%
    group_by(N.test) %>%
    summarise(power=mean(sig))
```

N.test	power
200	0.568
300	0.733

Of course, in the case of a single predictor, one can determine an analytic solution to this:

```
# f2 = R^2 / (1 - R^2)
pwr.f2.test(u=1, v=c(200-2, 300-2), f2=(.15^2) / (1 - .15^2))
#>
#>      Multiple regression power calculation
#>
#>               u = 1
#>               v = 198, 298
#>              f2 = 0.0230179
#>       sig.level = 0.05
#>           power = 0.5695666, 0.7451719
```

However, using simulation, we can determine the power for more complex models, including interactions and simple effects.

```
lm_test_interaction <- function(simNum, N, b1, b2, b3, b0=0, x1m=0, x1sd=1,
    x2m=0, x2sd=1) {

    x1 <- rnorm(N, x1m, x1sd)
    x2 <- rnorm(N, x2m, x2sd)
    yvar <- sqrt(1 - b1^2 - b2^2 - b3^2)  # residual variance
    y <- rnorm(N, b0 + b1*x1 + b2*x2 + b3*x1*x2, yvar)
    model <- lm(y ~ x1 * x2)

    # pull output from model (two main effects and interaction)
    est_x1 <- coef(summary(model))['x1', 'Estimate']
    p_x1 <- coef(summary(model))['x1', 'Pr(>|t|)']
    sig_x1 <- p_x1 < .05
    est_x2 <- coef(summary(model))['x2', 'Estimate']
    p_x2 <- coef(summary(model))['x2', 'Pr(>|t|)']
    sig_x2 <- p_x2 < .05
```

```
    est_int <- coef(summary(model))['x1:x2', 'Estimate']
    p_int <- coef(summary(model))['x1:x2', 'Pr(>|t|)']
    sig_int <- p_int < .05

    return(c(est_x1=est_x1, p_x1=p_x1, sig_x1=sig_x1, est_x2=est_x2, p_x2=p_x2,
        sig_x2=sig_x2, est_int=est_int, p_int=p_int, sig_int=sig_int))
}

# varying N at 200 and 300; setting coefficient of x1 = .15, coefficient of
# x2 = 0, and coefficien of interaction = .3
power_lm_int <- grid_search(lm_test_interaction, params=list(N=c(200, 300)),
    n.iter=5000, output='data.frame', b1=.15, b2=0, b3=.3, parallel='snow', ncpus=4)
results(power_lm_int) %>%
    group_by(N.test) %>%
    summarise(
        power_x1=mean(sig_x1),
        power_x2=mean(sig_x2),
        power_int=mean(sig_int))
```

N.test	power_x1	power_x2	power_int
200	0.593	0.052	0.988
300	0.774	0.051	0.999

Here, we are able to calculate the power for three coefficients at the same time. Note that for the coefficient b2, even though we set the true parameter equal to 0, power will trend toward your alpha level (typically .05), as this is the rate of false positives.

We can also calculate the power for simple effects:

```
lm_test_simple <- function(simNum, N, b1, b2, b3, b0=0, x1m=0, x1sd=1, x2m=0, x2sd=1) {
    x1 <- rnorm(N, x1m, x1sd)
    x2 <- rnorm(N, x2m, x2sd)
    yvar <- sqrt(1 - b1^2 - b2^2 - b3^2)
    y <- rnorm(N, b0 + b1*x1 + b2*x2 + b3*x1*x2, yvar)
    model <- lm(y ~ x1 * x2)  # here is the original model

    est_int <- coef(summary(model))['x1:x2', 'Estimate']
    p_int <- coef(summary(model))['x1:x2', 'Pr(>|t|)']
    sig_int <- p_int < .05

    # calculate x1 at +/- 1 SD, to look at simple effects
    x1minus1sd <- x1 - mean(x1) + sd(x1)
    x1plus1sd <- x1 - mean(x1) - sd(x1)

    # new models to examine simple effects
    model2 <- lm(y ~ x1minus1sd * x2)
    model3 <- lm(y ~ x1plus1sd * x2)

    # test effect of x2 when x1 is at +/- 1 SD
    est_x2_minus1 <- coef(summary(model2))['x2', 'Estimate']
    p_x2_minus1 <- coef(summary(model2))['x2', 'Pr(>|t|)']
    sig_x2_minus1 <- p_x2_minus1 < .05
```

```
    est_x2_plus1 <- coef(summary(model3))['x2', 'Estimate']
    p_x2_plus1 <- coef(summary(model3))['x2', 'Pr(>|t|)']
    sig_x2_plus1 <- p_x2_plus1 < .05

    return(c(est_int=est_int, p_int=p_int, sig_int=sig_int,
        est_x2_minus1=est_x2_minus1, p_x2_minus1=p_x2_minus1,
        sig_x2_minus1=sig_x2_minus1, est_x2_plus1=est_x2_plus1,
        p_x2_plus1=p_x2_plus1, sig_x2_plus1=sig_x2_plus1))
}

power_lm_simple <- grid_search(lm_test_simple, params=list(N=c(200, 300)),
    n.iter=5000, output='data.frame', b1=.15, b2=0, b3=.3, parallel='snow', ncpus=4)
results(power_lm_simple) %>%
    group_by(N.test) %>%
    summarise(
        power_x2_minus1=mean(sig_x2_minus1),
        power_x2_plus1=mean(sig_x2_plus1))
```

N.test	power_x2_minus1	power_x2_plus1
200	0.862	0.866
300	0.964	0.968

Multilevel models

Multilevel models (MLM) can be especially difficult to estimate power for, as there are numerous parameters that can vary. In the example below, we examine a simple MLM examining the effect of time on a variable measured at 4 time points.

```
mlm_test <- function(simNum, N, b1, b0=0, xm=0, xsd=1, varInt=1, varSlope=1, varResid=1) {
    timePoints <- 4
    subject <- rep(1:N, each=timePoints)
    sub_int <- rep(rnorm(N, 0, sqrt(varInt)), each=timePoints)  # random intercept
    sub_slope <- rep(rnorm(N, 0, sqrt(varSlope)), each=timePoints)  # random slope
    time <- rep(0:(timePoints-1), N)
    y <- (b0 + sub_int) + (b1 + sub_slope)*time + rnorm(N*timePoints, 0, sqrt(varResid))
        # y-intercept as a function of b0 plus random intercept;
        # slope as a function of b1 plus random slope
    data <- data.frame(subject, sub_int, sub_slope, time, y)

    # for more complex models that might not converge, tryCatch() is probably
    # a good idea
    return <- tryCatch({
        model <- nlme::lme(y ~ time, random=~time|subject, data=data)
            # when using parallel processing, we must refer to functions from
            # packages directly, e.g., package::function()

        est <- summary(model)$tTable['time', 'Value']
        se <- summary(model)$tTable['time', 'Std.Error']
        p <- summary(model)$tTable['time', 'p-value']
```

```
        return(c(est=est, se=se, p=p, sig=(p < .05)))
    },
    error=function(e) {
        #message(e)  # print error message
        return(c(est=NA, se=NA, p=NA, sig=NA))
    })

    return(return)
}

# I am cutting this down to 500 iterations so that the document compiles faster; I would, however,
# recommend more iterations for a stable estimate
power_mlm <- grid_search(mlm_test, params=list(N=c(200, 300)), n.iter=500, output='data.frame',
b1=.15,
    varInt=.05, varSlope=.15, varResid=.4, parallel='snow', ncpus=4)
results(power_mlm) %>%
    group_by(N.test) %>%
    summarise(
        power=mean(sig, na.rm=TRUE),
        na=sum(is.na(sig)))  # we use this to count up how many cases did not properly converge
```

N.test	power	na
200	0.993	65
300	1.000	32

Structural equation modelling

Here is an example of a simple mediation model:

```
med_test <- function(simNum, N, aa, bb, cc) {
    x <- rnorm(N, 0, 1)
    m <- rnorm(N, aa*x, sqrt(1 - aa^2))
    y <- rnorm(N, cc*x + bb*m, sqrt(1 - cc^2 - bb^2))
    data <- data.frame(x, m, y)

    # set up lavaan model to calculate indirect effect (ab) and total effect
    model <- '
        m ~ a*x
        y ~ c*x
        y ~ b*m
        ab := a*b
        total := c + (a*b)'

    fit <- lavaan::sem(model, data=data)
    ests <- lavaan::parameterEstimates(fit)
        # when using parallel processing, we must refer to functions from
        # packages directly, e.g., package::function()

    # pull output from model
    a_est <- ests[ests$label == 'a', 'est']
```

```
    a_p <- ests[ests$label == 'a', 'pvalue']
    b_est <- ests[ests$label == 'b', 'est']
    b_p <- ests[ests$label == 'b', 'pvalue']
    c_est <- ests[ests$label == 'c', 'est']
    c_p <- ests[ests$label == 'c', 'pvalue']
    ab_est <- ests[ests$label == 'ab', 'est']
    ab_p <- ests[ests$label == 'ab', 'pvalue']

    return(c(a_est=a_est, a_p=a_p, b_est=b_est, b_p=b_p, c_est=c_est, c_p=c_p,
        ab_est=ab_est, ab_p=ab_p, sig=(ab_p < .05)))
}

# set up mediation model where x -> m = .15, m -> y = .2, and x -> y = .05
power_med <- grid_search(med_test, params=list(N=c(200, 300)), n.iter=1000, output='data.frame',
aa=.15,
    bb=.2, cc=.05, parallel='snow', ncpus=4)
results(power_med) %>%
    group_by(N.test) %>%
    summarise(
        power=mean(sig, na.rm=TRUE))
```

N.test	power
200	0.238
300	0.482

And an example of predicting a latent variable:

```
latent_test <- function(simNum, N, b1, ind1, ind2, ind3) {
    # matrix of factor structure; we have x as observed predictor, and y is a
    # latent variable with three indicators
    fmodel <- matrix(
        c(1, 0,      # x
          0, ind1,   # y1
          0, ind2,   # y2
          0, ind3),  # y3
        nrow=4, ncol=2, byrow=TRUE, dimnames=list(
            c('x', 'y1', 'y2', 'y3'),  # rows (observed)
            c('x', 'y')))              # cols (latent)

    # matrix of effects structure (variance-covariance); we are using x to
    # predict y (with coefficient specified as b1)
    y_resid_var <- sqrt(1 - b1^2)
    effects <- matrix(
        c(1, b1,            # x
          0, y_resid_var),  # y
        nrow=2, ncol=2, byrow=TRUE, dimnames=list(
            c('x', 'y'), c('x', 'y')))

    data <- paramtest::gen_data(fmodel, effects)
        # generates the data using factor and effects matrices
```

```
    model <- '
        y =~ y1 + y2 + y3
        y ~ b1*x'

    fit <- lavaan::sem(model, data=data)
    ests <- lavaan::parameterEstimates(fit)

    est <- ests[ests$label == 'b1', 'est']
    p <- ests[ests$label == 'b1', 'pvalue']

    return(c(est=est, p=p, sig=(p < .05)))
}

power_sem <- grid_search(latent_test, params=list(N=c(200, 300)), n.iter=1000, output='data.frame',
    b1=.15, ind1=.4, ind2=.4, ind3=.4, parallel='snow', ncpus=4)
results(power_sem) %>%
    group_by(N.test) %>%
    summarise(
        power=mean(sig, na.rm=TRUE))
```

N.test	power
200	0.802
300	0.801

Summary

Simulating the statistical power of complex models can be challenging due to the number of parameters that one needs to estimate. Making assumptions about how the variables covary, how they relate to each other, etc. can make it difficult. However, using the ‘paramtest’ package provides a flexible way to run simulations in order to properly estimate power across a variety of assumptions. Hopefully, the examples in this vignette can provide you with a useful template from which to create models that fit your particular needs.