A Quick Guide to Conducting a Simulation Study

Luke Miratrix

2017-04-18 17:59:53

Overview

The code in this document is a series of case studies of simple simulation studies where we examine such things as the behavior of the simple difference-in-means for detecting treatment effect in a randomized experiment.

The primary purpose of this document is to illustrate how one might generate functions to conduct a simulation given a specific set of parameters, and then build on such functions to explore a range of parameter settings in what we would call multi-factor simulation experiments.

This script shows how you can streamline this using a few useful R functions in order to get nice and tidy code with nice and tidy simulation results.

The first simulation study presented looks at the t-test under violations of the normality assumption. The second is, essentially, a power analysis. Here we have a single estimator and we are evaluating how it works in a variety of cicumstances. In the third simulation we compare different estimators via simulation by showing a simulation comparing the mean, trimmed mean and median for estimating the center of a distribution.

This script relies on the `tidyverse` package, which needs to be loaded.

```
library( tidyverse )
```

We use methods from the "tidyverse" for cleaner code and some nice shortcuts. (see the R for Data Science textbook for more on this).

Technical note: This script was compiled from a simple `.R` file. If you are reading the raw file you will notice some "`#+`" which indicate directives to R code blocks (chunks) in the knitr (R markdown) world. This impacts how the code is run and displayed when turning the R file into a pdf to read. The comments beginning with "`#'`" are interpreted as markdown when the document is complied in RStudio as a notebook via `knitr:spin()` to make the pretty tutorial pdf.

Simulation 1: the performance of the t-test

We will start with a simulation study to examine the coverage of the lowly one-sample t-test. *Coverage* is the chance of a confidence interval capturing the true parameter value. Let's first look at our test on some fake data:

```
# make fake data
dat = rnorm( 10, mean=3, sd=1 )

# conduct the test
tt = t.test( dat )
tt
```

```
##
##  One Sample t-test
##
## data:  dat
## t = 10.134, df = 9, p-value = 3.202e-06
## alternative hypothesis: true mean is not equal to 0
```

```
## 95 percent confidence interval:
##  2.208874 3.478401
## sample estimates:
## mean of x
##  2.843637
```

```
# examine the results
tt$conf.int
```

```
## [1] 2.208874 3.478401
## attr(,"conf.level")
## [1] 0.95
```

For us, we have a true mean of 3. Did we capture it? To find out, we use `findInterval()`

```
findInterval( 3, tt$conf.int )
```

```
## [1] 1
```

`findInterval()` checks to see where the first number lies relative to the range given in the second argument. E.g.,

```
findInterval( 1, c(20, 30) )
```

```
## [1] 0
```

```
findInterval( 25, c(20, 30) )
```

```
## [1] 1
```

```
findInterval( 40, c(20, 30) )
```

```
## [1] 2
```

So, for us, `findInterval == 1` means we got it! Packaging the above gives us the following code:

```
# make fake data
dat = rnorm( 10, mean=3, sd=1 )

# conduct the test
tt = t.test( dat )
tt
```

```
##
##  One Sample t-test
##
## data:  dat
## t = 7.7336, df = 9, p-value = 2.898e-05
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  1.808456 3.303855
## sample estimates:
## mean of x
##  2.556156
```

```
# evaluate the results
findInterval( 3, tt$conf.int ) == 1
```

```
## [1] TRUE
```

The above shows the canonical form of a single simulation trial: make the data, analyze the data, decide how well we did.

Now let's look at coverage by doing the above many, many times and seeing how often we capture the true parameter:

```
rps = replicate( 1000, {
    dat = rnorm( 10 )
    tt = t.test( dat )
    findInterval( 0, tt$conf.int )
})
table( rps )
```

```
## rps
##   0   1   2
##  21 953  26
```

```
mean( rps == 1 )
```

```
## [1] 0.953
```

We got about 95% coverage, which is good news. We can also assess *simulation uncertainty* by recognizing that our simulation results are an i.i.d. sample of the infinite possible simulation runs. We analyze this sample to see a range for our true coverage.

```
hits = as.numeric( rps == 1 )
prop.test( sum(hits), length(hits), p = 0.95 )
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  sum(hits) out of length(hits), null probability 0.95
## X-squared = 0.13158, df = 1, p-value = 0.7168
## alternative hypothesis: true p is not equal to 0.95
## 95 percent confidence interval:
##  0.9374968 0.9649054
## sample estimates:
##     p
## 0.953
```

We have no evidence that our coverage is not what it should be: 95%.

Things working out should hardly be surprising. The t-test is designed for normal data and we generated normal data. In other words, our test is following theory when we meet our assumptions. Now let's look at an exponential distribution to see what happens when we don't have normally distributed data. We are simulating to see what happens when we voilate our assumptions behind the t-test. Here, the true mean is 1 (the mean of a standard exponential is 1).

```
rps = replicate( 1000, {
    dat = rexp( 10 )
    tt = t.test( dat )
    findInterval( 1, tt$conf.int )
})
table( rps )
```

```
## rps
##   0   1   2
##   6 899  95
```

Our interval is often entirely too low and very rarely does our interval miss because it is entirely too high. Furthermore, our average coverage is not 95% as it should be:

```
mean( rps == 1 )
```

```
## [1] 0.899
```

Again, to take simulation uncertainty into account we do a proportion test. Here we have a confidence interval of our true coverage under our model misspecification:

```
hits = as.numeric( rps == 1 )
prop.test( sum(hits), length(hits) )
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  sum(hits) out of length(hits), null probability 0.5
## X-squared = 635.21, df = 1, p-value < 2.2e-16
## alternative hypothesis: true p is not equal to 0.5
## 95 percent confidence interval:
##  0.8782316 0.9166333
## sample estimates:
##     p
## 0.899
```

Our coverage is *too low*. Our t-test based confidence interval is missing the true value (1) more than it should.

Finally, we want to examine how the coverage changes as the sample size varies. So let's do a one-factor experiment, with the factor being sample size. I.e., we will conduct the above simulation for a variety of sample sizes and see how coverage changes.

We first make a function, wrapping up our *specific, single-scenario* simulation into a bundle so we can call it under a variety of different scenarios.

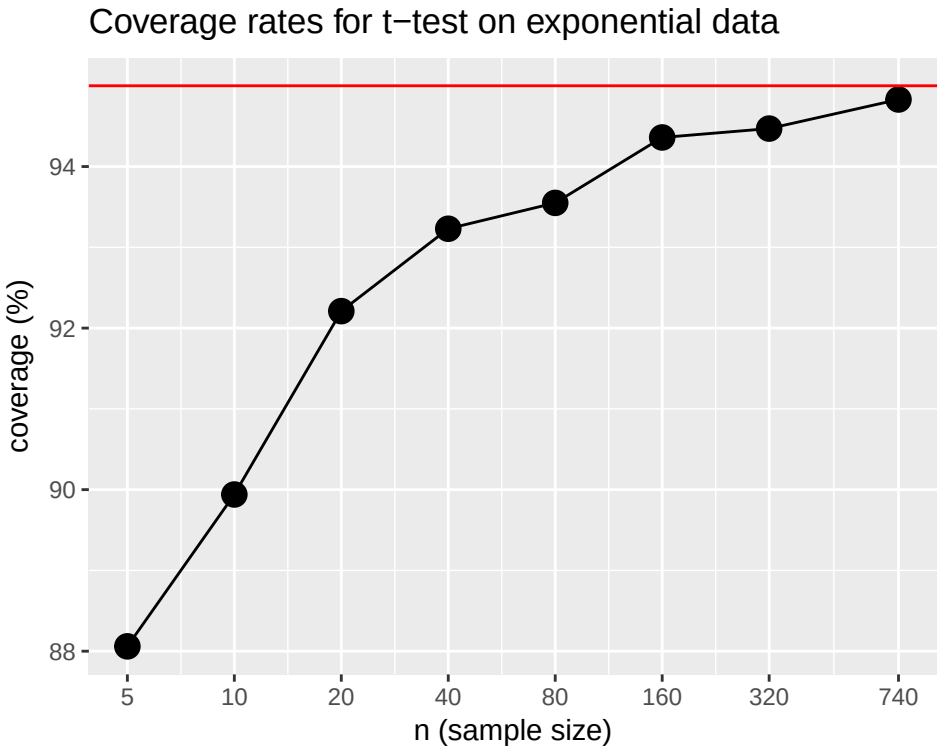
```
run.experiment = function( n ) {
    rps = replicate( 10000, {
        dat = rexp( n )
        tt = t.test( dat )
        findInterval( 1, tt$conf.int )
    })

    mean( rps == 1 )
}
```

Now we run `run.experiment` for different n. We do this with `map_dbl()`, which takes a list and calls a function for each value in the list (See R for DS, Chapter 21.5).

```
ns = c( 5, 10, 20, 40, 80, 160, 320, 740 )
cover = map_dbl( ns, run.experiment )
```

Make a data.frame of our results and plot:

```
res = data.frame( n = ns, coverage=cover )
ggplot( res, aes( x=n, y=100*coverage ) ) +
    geom_line() + geom_point( size=4 ) +
    geom_hline( yintercept=95, col="red" ) +
    scale_x_log10( breaks=ns ) +
    labs( title="Coverage rates for t-test on exponential data",
          x = "n (sample size)", y = "coverage (%)" )
```



Note the plot is on a log scale for the x-axis.

So far we have done a very simple simulation to assess how well a statistical method works in a given circumstance. We have run a single factor experiment, systematically varying the sample size to examine how the behavior of our estimator changes. In this case, we find that coverage is poor for small sample sizes, and still a bit low for higher sample sizes is well. The overall framework is to repeatidly do the following:

- Generate data according to some decided upon data generation process (DGP). This is our model.
- Analyze data according to some other process (and possibly some other assumed model).
- Assess whether the analysis "worked" by some measure of working (such as coverage).

We next extend this general simulation framework to look at how to very multiple things at once. This is called a multifactor experiment.

Simulation 2: The Power and Validity of Neyman's ATE Estimate

The way to build a simulation experiment is to first write code to run a specific simulation for a specific scenario. Once that is working, we will re-use the code to systematically explore a variety of scenarios so we can see how things change as scenario changes. Next I would build up this system.

For our running example we are going to look at a randomized experiment. We will assume the treatment and control groups are normally distributed with two different means. We will generate a random data set, estimate the treatment effect by taking the difference in means and calculating the associated standard error, and generating a p-value using the normal approximation. (As we will see, this is not a good idea for small sample size since we should be using a t-test style approach.)

Step 1: Write a function for a specific simulation given specific parameters.

Our function will generate two groups of the given sizes, one treatment and one control, and then calculate the difference in means. It will then test this difference using the normal approximation.

The function also calculates and returns the effect size as the treatment effect divided by the control standard deviation (useful for understanding power, shown later on).

```
run.one = function( nC, nT, sd, tau, mu = 5 ) {
    Y0 = mu + rnorm( nC, sd=sd )
    Y1 = mu + tau + rnorm( nT, sd=sd )

    tau.hat = mean( Y1 ) - mean( Y0 )
    SE.hat = sqrt( var( Y0 ) / ( nC ) + var( Y1 ) / ( nT ) )

    z = tau.hat / SE.hat
    pv = 2 * (1 - pnorm( abs( z ) ))

    c( tau.hat = tau.hat, ES = tau / sd, SE.hat = SE.hat, z=z, p.value=pv )
}
```

A single run will generate a data set, analyze it, and give us back a variety of results as a list.

```
run.one( nT=5, nC=10, sd=1, tau=0.5 )
```

```
##   tau.hat        ES    SE.hat         z   p.value
## 0.3411543 0.5000000 0.5552000 0.6144709 0.5389042
```

Running our single trial more than once

The following code borrows a useful function from the older plyr package (you may need to install it). It is a version of `replicate()` that runs a chunk of code a given number of times. The difference is that `rdply` returns everything as a data frame! Sweet!

```
eres <- plyr::rdply( 500, run.one( nC=10, nT=10, sd=1, tau=0.5 ) )

# Each row is a simulation run:
head( eres )
```

```
##   .n    tau.hat  ES    SE.hat          z     p.value
## 1  1  0.8982500 0.5 0.4151147  2.1638596 0.030475129
## 2  2  1.1968364 0.5 0.3728216  3.2102119 0.001326371
## 3  3  0.6681853 0.5 0.5040343  1.3256741 0.184947662
## 4  4  0.7083930 0.5 0.3494866  2.0269532 0.042667203
## 5  5 -0.1082389 0.5 0.4858170 -0.2227976 0.823693047
## 6  6  0.6333932 0.5 0.4425515  1.4312304 0.152364186
```

We then summarize our results with the `dplyr summarise` function. Our summarization calculates the average treatment effect estimate `E.tau.hat`, the average Standard Error estimate `E.SE.hat`, the average Effect Size `ES`, and the power `power` (defined as the percent of time we reject at alpha=0.05, i.e., the percent of times our p-value was less than our 0.05 threshold):

```
eres %>% summarise( E.tau.hat = mean( tau.hat ),
                    E.SE.hat = mean( SE.hat ),
                    ES = mean( ES ),
                    power = mean( p.value <= 0.05 ) )
```

```
##   E.tau.hat  E.SE.hat  ES power
## 1 0.4779893 0.4441824 0.5 0.192
```

We bundle the above into a function that runs our single trial multiple times and summarizes the results:

```
run.experiment = function( nC, nT, sd, tau, mu = 5, R = 500 ) {

    eres = plyr::rdply( R, run.one( nC, nT, sd, tau, mu ) )
    eres %>% summarise( E.tau.hat = mean( tau.hat ),
                        E.SE.hat = mean( SE.hat ),
                        ES = mean( ES ),
                        power = mean( p.value <= 0.05 ) ) %>%
        mutate( nC=nC, nT=nT, sd=sd, tau=tau, mu=mu, R=R )
}
```

Our function also adds in the details of the simulation (the parameters we passed to the `run.one()` call).

Test our function to see what we get:

```
run.experiment( 10, 3, 1, 0.5 )
```

```
##   E.tau.hat  E.SE.hat  ES power nC nT sd tau mu   R
## 1 0.5203716 0.6172025 0.5  0.21 10  3  1 0.5  5 500
```

Key point: We want a dataframe back from `run.experiment()`, because, after calling `run.experiment()` many times, we are going to stack the results up to make one long dataframe of results. Happily the `dplyr` package gives us dataframes so this is not a problem here.

Step 2: Make a dataframe of all experimental combinations desired

We use the above to run a *multi-factor simulation experiment*. We are going to vary four factors: control group size, treatment group size, standard deviation of the units, and the treatment effect.

We first set up the levels we want to have for each of our factors (these are our *simulation parameters*).

```
nC = c( 2, 4, 7, 10, 50, 500 )
nT = c( 2, 4, 7, 10, 50, 500 )
sds = c( 1, 2 )
tau = c( 0, 0.5, 1 )
```

We then, using `expand.grid()` generate a dataframe of all combinations of our factors.

```
experiments = expand.grid( nC=nC, nT=nT, sd=sds, tau=tau )
head( experiments )
```

```
##    nC nT sd tau
## 1   2  2  1   0
## 2   4  2  1   0
## 3   7  2  1   0
## 4  10  2  1   0
## 5  50  2  1   0
## 6 500  2  1   0
```

See what we get? One row will correspond to a single experimental run. Note how the parameters we would pass to `run.experiment()` correspond to the columns of our dataset.

Also, is easy to end up running a lot of experiments!

```
nrow( experiments )
```

```
## [1] 216
```

We next run an experiment for each row of our dataframe of experiment factor combinations using the `pmap_df()` function which will, for each row in our dataframe, call `run.experiment()`, passing one parameter taken from each column of our dataframe.

```
exp.res <- experiments %>% pmap_df( run.experiment, R=500 )
```

The `R=500` after `run.experiment` passes the *same* parameter of $R = 500$ to each run (we run the same number of trials for each experiment).

Here is a peek at our results:

```
head( exp.res )
```

```
##     E.tau.hat  E.SE.hat ES power  nC nT sd tau mu   R
## 1 -0.10669053 0.8583547  0 0.178   2  2  1   0  5 500
## 2  0.05035076 0.7827624  0 0.188   4  2  1   0  5 500
## 3 -0.02966955 0.7103727  0 0.190   7  2  1   0  5 500
## 4  0.02488272 0.6580385  0 0.200  10  2  1   0  5 500
## 5 -0.02023249 0.5773534  0 0.250  50  2  1   0  5 500
## 6  0.04848068 0.5382369  0 0.324 500  2  1   0  5 500
```

At this point you should save your simulation results to a file. This is especially true if the simulation happens to be quite time-intensive to run. Usually a csv file is sufficient.

We save using the tidyverse writing command; see "R for Data Science" textbook, 11.5.

```
write_csv( exp.res, "simulation_results.csv" )
```

Step 3: Explore results

Once your simulation is run, you want to evaluate the results. One would often put this code into a seperate '.R' file that loads this saved file to start. This allows for easily changing how one analyzes an experiment without re-running the entire thing.

Visualizing experimental results

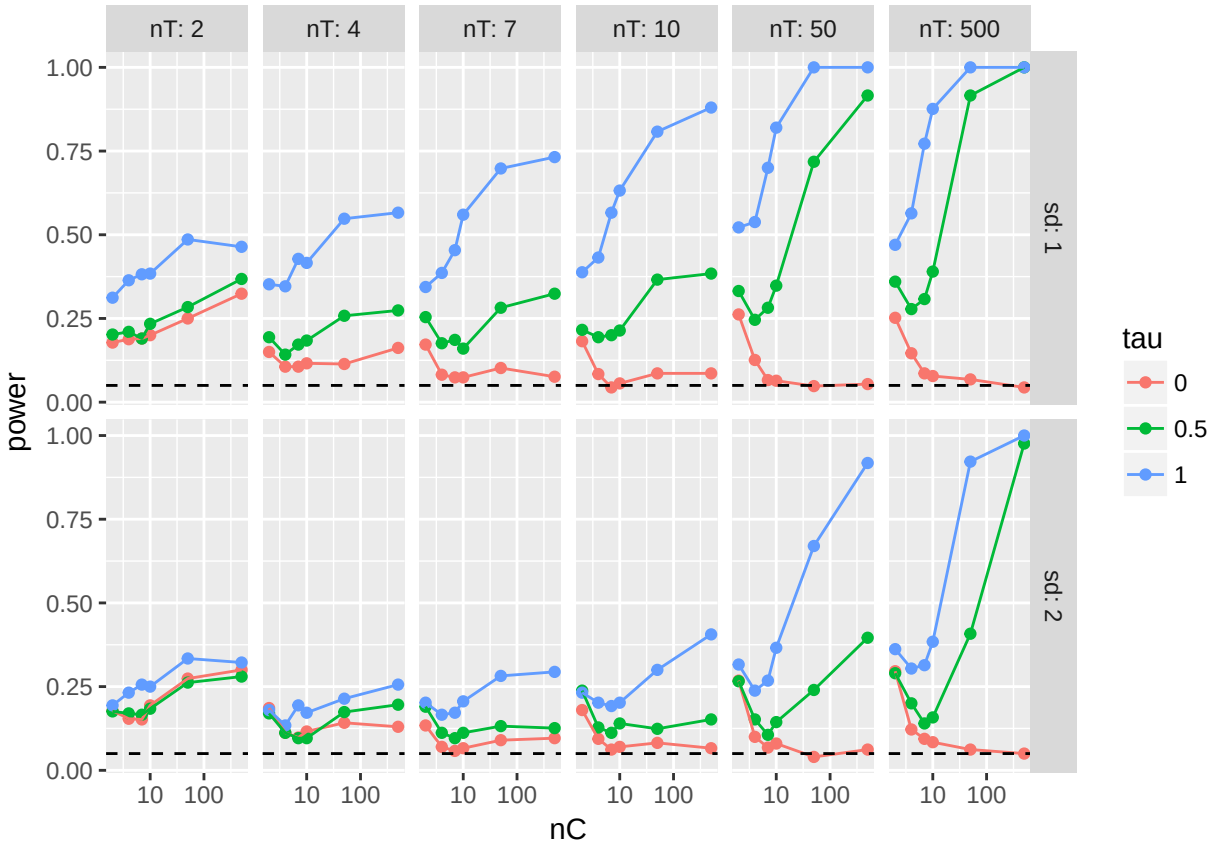
Plotting is always a good way to vizualize simulation results. Here we make our tau and ES into factors, so `ggplot` behaves, and then plot all our experiments as two rows based on one factor (`sd`) with the columns being another (`nT`). (This style of plotting a bunch of small plots is called "many multiples" and is beloved by Tufte.) Within each plot we have the x-axis for one factor (`nC`) and multiple lines for the final factor (`tau`). The y-axis is our outcome of interest, power. We add a 0.05 line to show when we are rejecting at rates above our nominal α. This plot shows the relationship of 5 variables.

```
exp.res = read_csv( "simulation_results.csv" )
```

```
## Parsed with column specification:
## cols(
##   E.tau.hat = col_double(),
##   E.SE.hat = col_double(),
##   ES = col_double(),
##   power = col_double(),
##   nC = col_double(),
##   nT = col_double(),
##   sd = col_double(),
##   tau = col_double(),
##   mu = col_double(),
```

```
##   R = col_double()
## )
```

```
exp.res = exp.res %>% mutate( tau = as.factor( tau ),
                              ES = as.factor( ES ) )
ggplot( exp.res, aes( x=nC, y=power, group=tau, col=tau ) ) +
    facet_grid( sd ~ nT, labeller=label_both ) +
    geom_point() + geom_line() +
    scale_x_log10() +
    geom_hline( yintercept=0.05, col="black", lty=2)
```



Note: We are seeing elevated rejection rates under the null for small and even moderate sample size! We can zoom in on specific simulations run, to get some more detail such as estimated power under the null for larger groups. Here we check and we are seeing rejection rates of around 0.05, which is what we want.

```
filter( exp.res, tau==0, nT >= 50, nC >= 50 )
```

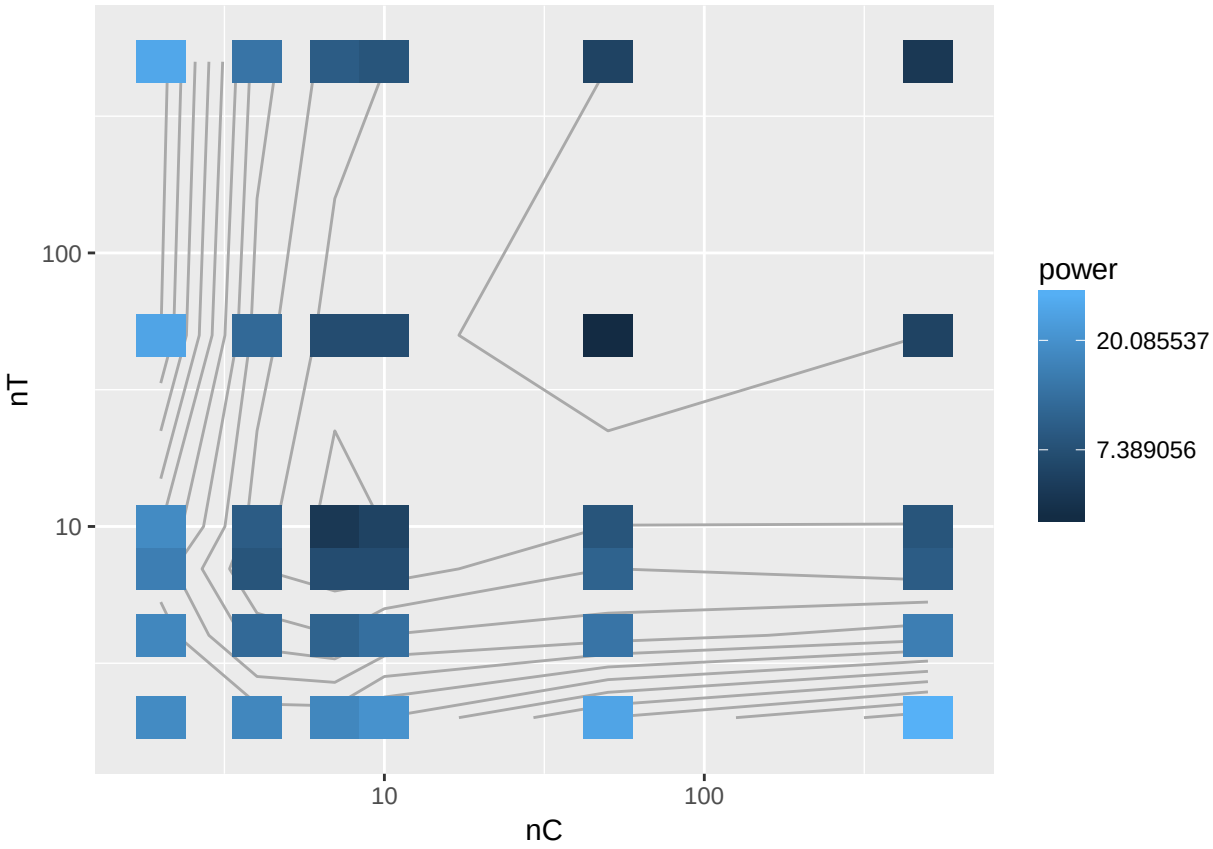
```
## # A tibble: 8 × 10
##      E.tau.hat   E.SE.hat     ES power    nC    nT    sd   tau    mu
##          <dbl>      <dbl> <fctr> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl>
## 1 -0.010968634 0.20015520      0 0.048    50    50     1      0     5
## 2 -0.002663422 0.14684854      0 0.054   500    50     1      0     5
## 3  0.005843591 0.14769551      0 0.068    50   500     1      0     5
## 4  0.001324795 0.06325634      0 0.044   500   500     1      0     5
## 5 -0.018673192 0.39954294      0 0.040    50    50     2      0     5
## 6 -0.010723597 0.29481583      0 0.062   500    50     2      0     5
## 7 -0.006330527 0.29459019      0 0.062    50   500     2      0     5
## 8 -0.005772006 0.12643916      0 0.050   500   500     2      0     5
```

```
## # ... with 1 more variables: R <dbl>
```

We can get fancy and look at rejection rate (power under `tau = 0`) as a function of both nC and nT using a contour-style plot:

```
exp.res.rej <- exp.res %>% filter( tau == 0 ) %>%
    group_by( nC, nT ) %>%
    summarize( power = mean( power ) )

exp.res.rej = mutate( exp.res.rej, power = round( power * 100 ) )
```

```
ggplot( filter( exp.res.rej ), aes( x=nC, y=nT ) ) +
    geom_contour( aes( z=power ), col="darkgrey" ) +
    scale_x_log10() + scale_y_log10() +
    geom_tile( aes( fill=power ) ) +
    scale_fill_gradient( trans="log")
```



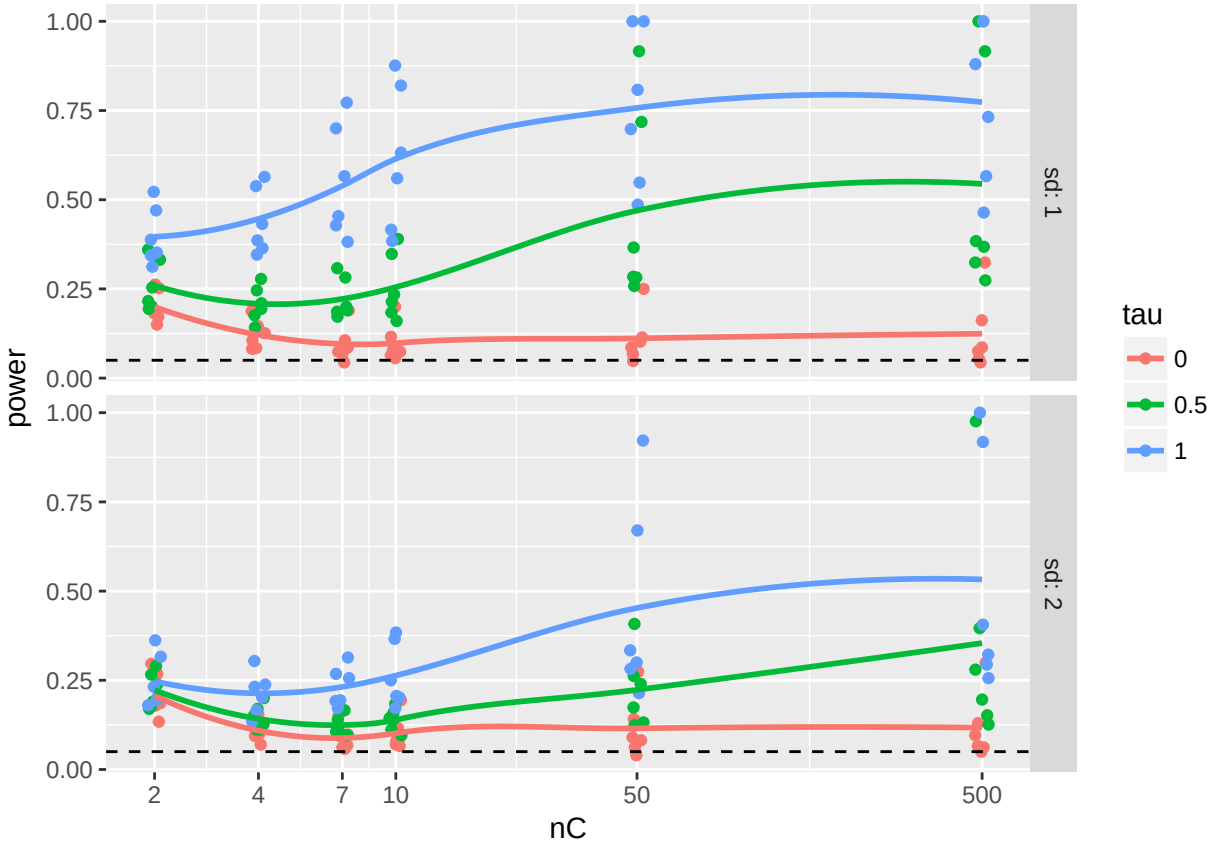
Admittidly, this plot needs some work to really show the areas with rejection rates of well above 0.05. But we see that small tx or co groups are both bad here.

Looking at main effects

We can ignore a factor and just look at another. This is looking at the **main effect** or **marginal effect** of the factor.

The easy way to do this is to let `ggplot` smooth our individual points on a plot. Be sure to also plot the individual points to see variation, however.

```
ggplot( exp.res, aes( x=nC, y=power, group=tau, col=tau ) ) +
    facet_grid( sd ~ ., labeller=label_both ) +
    geom_jitter( width=0.02, height=0 ) +
    geom_smooth( se = FALSE ) +
    scale_x_log10( breaks=nC) +
    geom_hline( yintercept=0.05, col="black", lty=2)
```



Note how we see our individual runs that we marginalize over.

To look at our main effects we can also summarize our results, averaging our experimental runs across other factor levels. For example, in the code below we average over the different treatment group sizes and standard deviations, and plot the marginalized results.

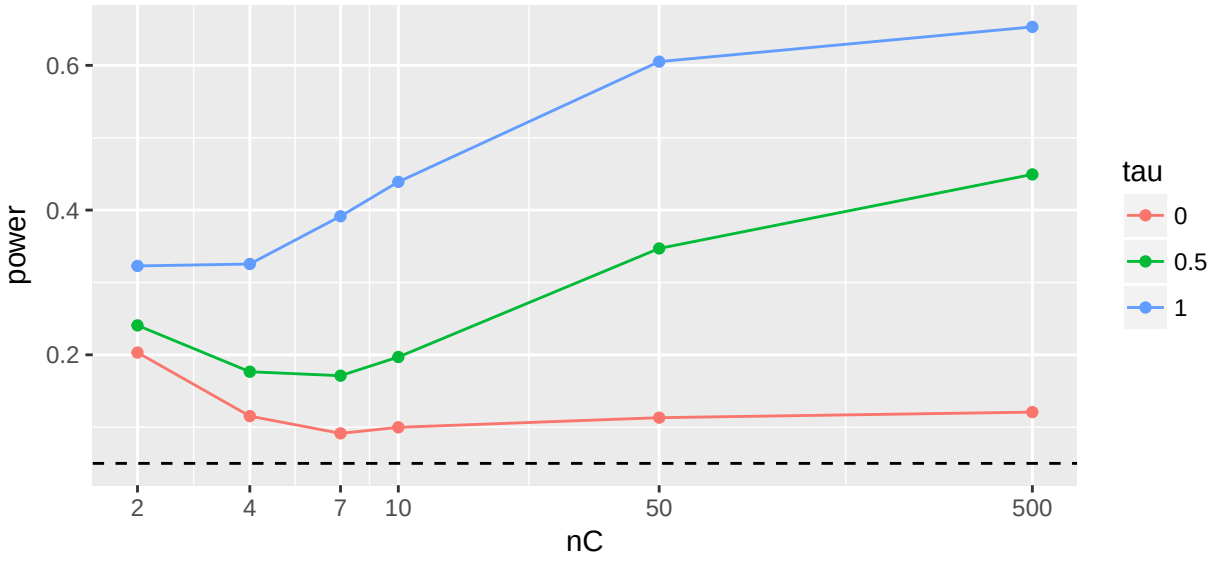
To marginalize, we group by the things we want to keep. `summarise()` then averages over the things we want to get rid of.

```
exp.res.sum = exp.res %>% group_by( nC, tau ) %>%
    summarise( power = mean( power ) )
head( exp.res.sum )
```

```
## Source: local data frame [6 x 3]
## Groups: nC [2]
##
##      nC    tau     power
##   <dbl> <fctr>     <dbl>
## 1     2      0 0.2031667
## 2     2    0.5 0.2406667
## 3     2      1 0.3228333
```

```
## 4     4      0 0.1153333
## 5     4    0.5 0.1766667
## 6     4      1 0.3255000
```

```
ggplot( exp.res.sum, aes( x=nC, y=power, group=tau, col=tau ) ) +
    geom_line() + geom_point() +
    scale_x_log10( breaks=nC) +
    geom_hline( yintercept=0.05, col="black", lty=2)
```

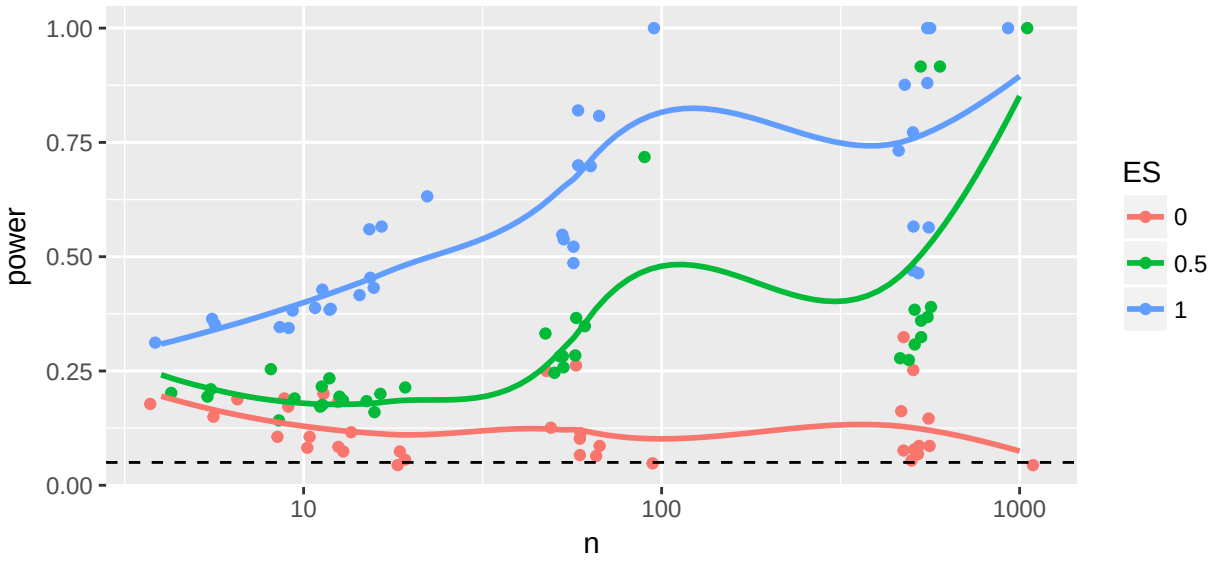


We can try to get clever and look at other aspects of our experimental runs. The above suggests that the smaller of the two groups is dictating things going awry, in terms of elevated rejection rates under the null. We can also look at things in terms of some other more easily interpretable parameter (here we switch to effect size instead of raw treatment effect).

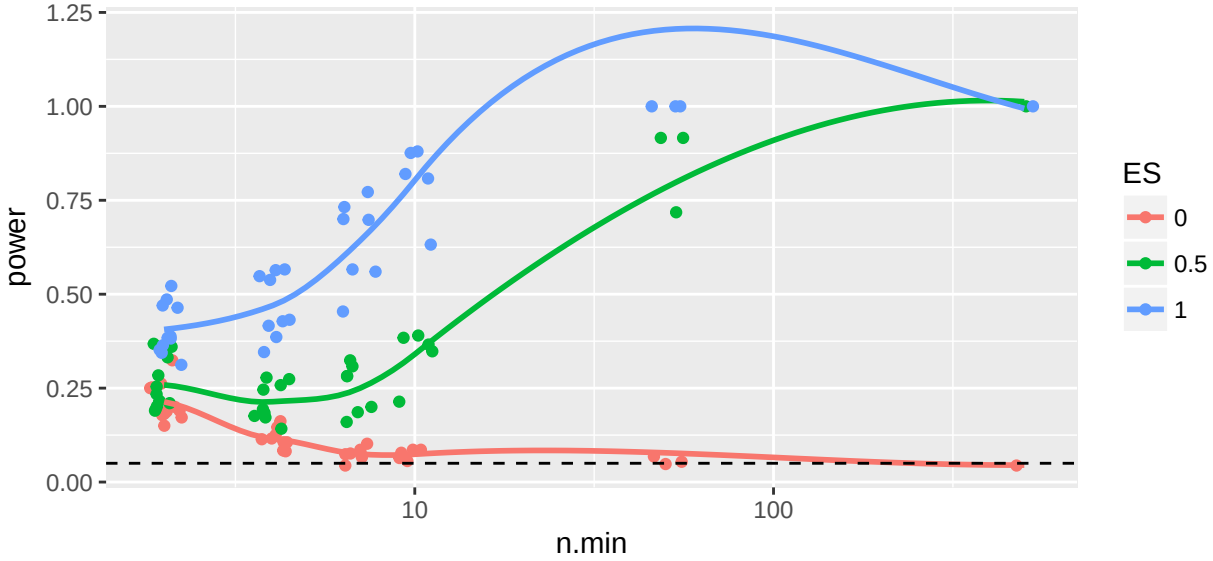
Given this, we might decide to look at total sample size or the smaller of the two groups sample size and make plots that way (we are also subsetting to just the `sd=1` cases as there is nothing interesting in both, really):

```
exp.res <- exp.res %>% mutate( n = nC + nT,
                               n.min = pmin( nC, nT ) )
```

```
ggplot( filter( exp.res, sd==1 ), aes( x=n, y=power, group=ES, col=ES ) ) +
    geom_jitter( width=0.05, height=0) +
    geom_smooth( se = FALSE) +
    scale_x_log10() +
    geom_hline( yintercept=0.05, col="black", lty=2)
```




```
ggplot( filter( exp.res, sd==1 ), aes( x=n.min, y=power, group=ES, col=ES ) ) +
    geom_jitter( width=0.05, height=0) +
    geom_smooth( se = FALSE) +
    scale_x_log10() +
    geom_hline( yintercept=0.05, col="black", lty=2)
```



Note the few observations out in the high `n.min` region for the second plot—this plot is a bit strange in that the different levels along the x-axis are assymetric with respect to each other. It is not balanced.

Addendum: Saving more details

Our `exp.res` dataframe from above has all our simulations, one simulation per row, with our measured outcomes. This is ideally all we need to analyze.

That being said, sometimes we might want to use a lot of disk space and keep much more. In particular, each row of `exp.res` corresponds to the summary of a whole collection of individual runs. We might instead store all of these runs.

To do this we just take the summarizing step out of our `run.experiment()`

```
run.experiment.raw = function( nC, nT, sd, tau, mu = 5, R = 500 ) {
    eres = plyr::rdply( R, run.one( nC, nT, sd, tau, mu ) )
    eres <- mutate( eres, nC=nC, nT=nT, sd=sd, tau=tau, mu=mu, R=R )
    eres
}
```

Each call to `run.experiment.raw()` gives one row per run. We replicate our simulation parameters for each row.

```
run.experiment.raw( 10, 3, 1, 0.5, R=4 )
```

```
##   .n      tau.hat  ES    SE.hat             z    p.value nC nT sd tau mu R
## 1  1 -0.450945068 0.5 0.5721977 -0.7880931360 0.43064223 10  3  1 0.5  5 4
## 2  2  0.000324705 0.5 0.3887987  0.0008351493 0.99933365 10  3  1 0.5  5 4
## 3  3  0.978188814 0.5 0.5914475  1.6538895212 0.09814996 10  3  1 0.5  5 4
## 4  4  0.138354391 0.5 0.6345716  0.2180280170 0.82740728 10  3  1 0.5  5 4
```

The advantage of this is we can then generate new outcome measures, as they occur to us, later on. The disadvantage is this result file will be R times as many rows as the older file, which can get quite, quite large.

But disk space is cheap! Here we run the same experiment with our more complete storage. Note how the `pmap_df` stacks the multiple rows from each run, giving us everything nicely bundled up:

```
exp.res.full <- experiments %>% pmap_df( run.experiment.raw, R=500 )
head( exp.res.full )
```

```
##   .n    tau.hat ES    SE.hat          z     p.value nC nT sd tau mu   R
## 1  1 -2.1199724  0 0.7128445 -2.9739622 0.002939813  2  2  1   0  5 500
## 2  2 -0.4355020  0 1.1252865 -0.3870144 0.698745570  2  2  1   0  5 500
## 3  3 -0.9984963  0 0.6073760 -1.6439508 0.100186372  2  2  1   0  5 500
## 4  4  0.3668887  0 0.4823516  0.7606251 0.446881050  2  2  1   0  5 500
## 5  5 -0.1213203  0 0.2667031 -0.4548888 0.649189226  2  2  1   0  5 500
## 6  6 -0.7861555  0 0.6349178 -1.2382004 0.215641756  2  2  1   0  5 500
```

We end up with a lot more rows:

```
nrow( exp.res.full )
```

```
## [1] 108000
```

```
nrow( exp.res )
```

```
## [1] 216
```

We next save our results:

```
write_csv( exp.res.full, "simulation_results_full.csv" )
```

Compare the file sizes: one is several k, the other is around 12 megabytes.

```
file.size("simulation_results.csv") / 1024
```

```
## [1] 16.06738
```

```
file.size("simulation_results_full.csv") / 1024
```

```
## [1] 11860.98
```

Getting results ready for analysis

If we generated raw results then we need to collapse them by experimental run before analyzing our results so we can explore the trends across the experiments. We do this by borrowing the summarise code from inside `run.experiment()`:

```
exp.res.sum <- exp.res.full %>%
    group_by( nC, nT, sd, tau, mu ) %>%
    summarise( R = n(),
               E.tau.hat = mean( tau.hat ),
               SE = sd( tau.hat ),
               E.SE.hat = mean( SE.hat ),
               ES = mean( ES ),
               power = mean( p.value <= 0.05 ) )
```

Note how I added an extra estimation of the true SE, just because I could! This is an easier fix, sometimes, than running all the simulations again after changing the `run.experiment()` method.

The results of summarizing during the simulation vs. after as we just did leads to the same place, however, although the order of rows in our final dataset are different (and we have a tibble instead of a data.frame, a consequence of using the `tidyverse`, but this is not something to worry about):

```
head( exp.res.sum )
```

```
## Source: local data frame [6 x 11]
## Groups: nC, nT, sd, tau [6]
##
##      nC    nT    sd   tau    mu     R     E.tau.hat        SE  E.SE.hat
##   <dbl> <dbl> <dbl> <dbl> <dbl> <int>         <dbl>     <dbl>     <dbl>
## 1     2     2     1   0.0     5   500  0.0235374919 1.0419384 0.8832385
## 2     2     2     1   0.5     5   500  0.5097628502 0.9924804 0.8962110
## 3     2     2     1   1.0     5   500  1.0314230696 0.9998192 0.8840027
## 4     2     2     2   0.0     5   500 -0.0001394172 2.0251479 1.8025675
## 5     2     2     2   0.5     5   500  0.5423638414 2.0436671 1.8226802
## 6     2     2     2   1.0     5   500  1.0222488645 1.9164870 1.7779536
## # ... with 2 more variables: ES <dbl>, power <dbl>
```

```
nrow( exp.res.sum )
```

```
## [1] 216
```

```
nrow( exp.res )
```

```
## [1] 216
```

Simulation 3: Comparing Estimators for the Mean via Simulation

The above ideas readily extend to when we wish to compare different forms of estimator for estimating the same thing. We still generate data, evaluate it, and see how well our evaluation works. The difference is we now evaluate it multiple ways, storing how the different ways work.

For our simple working example we are going to compare estimation of the center of a symmetric distribution via mean, trimmed mean, and median (so the mean and median are the same).

We are going to break this down into lots of functions to show the general framework. This framework can readily be extended to more complicated simulation studies.

For our data-generation function we will use the scaled t-distribution so the standard deviation will always be 1 but we will have different fatness of tails (high chance of outliers):

```
gen.data = function( n, df0 ) {
    rt( n, df=df0 ) / sqrt( df0 / (df0-2) )
}
```

The variance of a t is $df/(df-2)$, so if we divide our observations by the square root of this, we will standardize them so they have unit variance. See, the standard deviation is 1 (up to random error, and as long as df0 > 2)!:

```
sd( gen.data( 100000, df0 = 3 ) )
```

```
## [1] 0.9934177
```

We next define the parameter we want (this, the mean, is what we are trying to estimate):

```
mu = 0
```

Our analysis methods bundled in a function. We return a vector of the three estimates:

```
analyze.data = function( data ) {
    mn = mean( data )
    md = median( data )
    mn.tr = mean( data, trim=0.1 )
    data.frame( mean = mn, trim.mean=mn.tr, median=md )
}
```

Let's test:

```
dt = gen.data( 100, 3 )
analyze.data( dt )
```

```
##          mean    trim.mean     median
## 1 -0.04496413 0.0008906163 0.05742946
```

To evaluate, do a bunch of times, and assess results. Let's start by looking at a specific case. We generate 1000 datasets of size 10, and estimate the center using our three different estimators.

```
raw.exps <- plyr::rdply( 1000, {
    dt = gen.data( n=10, df0=5 )
    analyze.data( dt )
} )
```

We now have 1000 estimates for each of our estimators:

```
head( raw.exps )
```

```
##   .n        mean   trim.mean      median
## 1  1  0.17112057  0.34004900  0.52930650
## 2  2  0.41843941  0.36629805  0.37020937
## 3  3  0.15337650  0.03130291 -0.06930059
## 4  4 -0.33214689 -0.24003774 -0.19326752
## 5  5 -0.02775366  0.06270554  0.04614414
## 6  6 -0.20154089 -0.13609502 -0.10153322
```

We then want to assess estimator performance for each estimator. We first write a function to calculate what we want from 1000 estimates:

```
estimator.quality = function( estimates, mu ) {
    RMSE = sqrt( mean( (estimates - mu)^2 ) )
    bias = mean( estimates - mu )
```

```
    c( RMSE=RMSE, bias=bias, SE=sd( estimates ) )
}

estimator.quality( raw.exps$mean, mu )
```

```
##        RMSE        bias          SE
## 0.317177176 0.006625113 0.317266650
```

We now borrow another oldie-but-goodie from the `plyr` package. The function `ldply` is a transforming function that takes a list (`l`) and for everything in the list calls a given function. It packs up the results as a dataframe (`d`). For the record, `llply` would return everything as a list, `ddply` would take a dataframe and return a dataframe, and so forth. For us, `raw.exps[-1]` is a list of vectors, i.e., all the columns (except the first) of our dataframe of simulations. (Remember that a dataframe is a list of the variables in the dataframe.)

```
plyr::ldply( raw.exps[-1], estimator.quality, mu = mu, .id = "estimator" )
```

```
##   estimator      RMSE        bias        SE
## 1      mean 0.3171772 0.006625113 0.3172666
## 2 trim.mean 0.2939671 0.004584475 0.2940784
## 3    median 0.3129517 0.006146037 0.3130479
```

Note the `mu = 0` line after estimator.quality. We can pass extra arguments to the function by putting them after the function name. The function will take as its first argument the elements from the `raw.exps[-1]` list.

Aside: There is probably a nice way to do this in the `dplyr` package, but I don't know what it is.

To continue, we pack up the above into a function, as usual. Our function takes our two parameters of sample size and degrees of freedom, and returns a data frame of results.

```
one.run = function( n, df0 ) {
    raw.exps <- plyr::rdply( 1000, {
        dt = gen.data( n=n, df0=df0 )
        analyze.data( dt )
    } )
    rs <- plyr::ldply( raw.exps[-1], estimator.quality, mu = 0, .id = "estimator" )

    rs
}
```

Our function will take our two parameters, run a simulation, and give us the results. We see here that none of our estimators are particularly biased and the trimmed mean has, possibly, the smallest RMSE, although it is a close call.

```
one.run( 10, 5 )
```

```
##   estimator      RMSE         bias        SE
## 1      mean 0.3178515 -0.005641335 0.3179605
## 2 trim.mean 0.2948322 -0.006689340 0.2949038
## 3    median 0.3240756 -0.020217814 0.3236062
```

Ok, now we want to see how sample size impacts our different estimators. If we also vary degrees of freedom we have a *three*-factor experiment, where one of the factors is our estimator itself. We are going to use a new clever trick. As before, we use `pmap()`, but now we store the entire dataframe of results we get back from our function in a new column of our original dataframe. See R for DS, Chapter 25.3. This trick works best if we have everything as a `tibble` which is basically a dataframe that prints a lot nicer and doesn't try to second-guess what you are up to all the time.

```
ns = c( 10, 50, 250, 1250 )
dfs = c( 3, 5, 15, 30 )
lvls = expand.grid( n=ns, df=dfs )

# So it stores our dataframe results in our lvls data properly.
lvls = as_tibble(lvls)

results <- lvls %>% mutate( results = pmap( lvls, one.run ) )
```

We have stored our results (a bunch of dataframes) in our main matrix of simulation runs.

```
head( results )
```

```
## # A tibble: 6 × 3
##       n    df              results
##   <dbl> <dbl>               <list>
## 1    10     3 <data.frame [3 × 4]>
## 2    50     3 <data.frame [3 × 4]>
## 3   250     3 <data.frame [3 × 4]>
## 4  1250     3 <data.frame [3 × 4]>
## 5    10     5 <data.frame [3 × 4]>
## 6    50     5 <data.frame [3 × 4]>
```

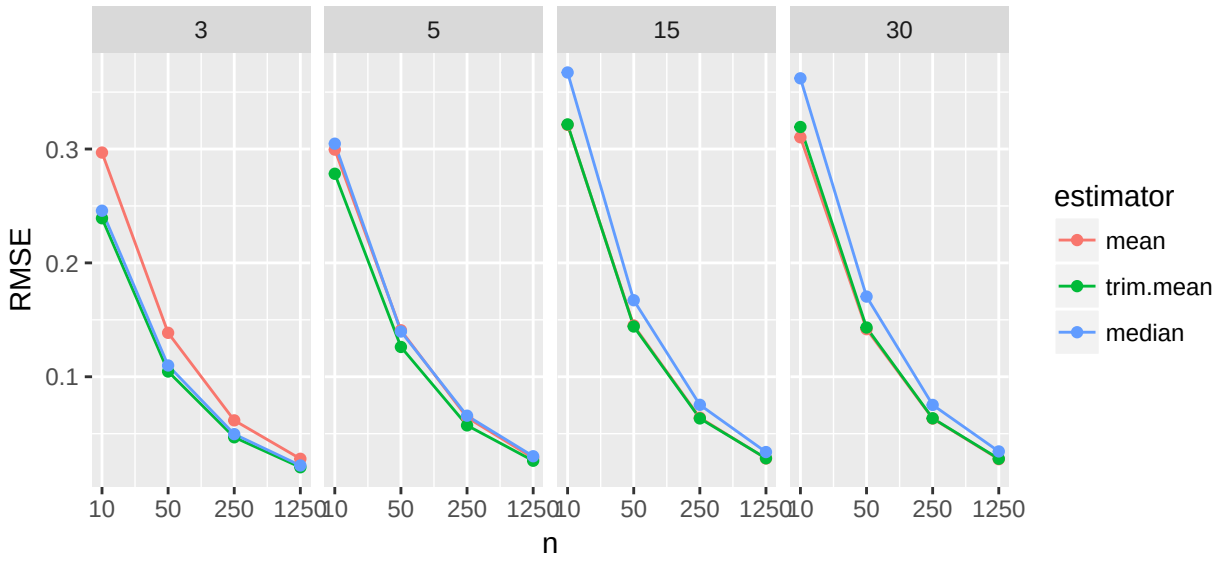
The unnest() function will unpack our dataframes and put everything together, all nice like. See (hard to read) R for DS Chapter 25.4.

```
results <- unnest( results )
results
```

```
## # A tibble: 48 × 6
##        n    df estimator       RMSE          bias         SE
##    <dbl> <dbl>    <fctr>      <dbl>         <dbl>      <dbl>
## 1     10     3      mean 0.29688802 -0.0068160151 0.29695829
## 2     10     3 trim.mean 0.23919464  0.0008088718 0.23931296
## 3     10     3    median 0.24581301 -0.0008059686 0.24593469
## 4     50     3      mean 0.13856862  0.0055025654 0.13852860
## 5     50     3 trim.mean 0.10460456  0.0029192629 0.10461614
## 6     50     3    median 0.10989340 -0.0014669735 0.10993859
## 7    250     3      mean 0.06169849 -0.0007136053 0.06172523
## 8    250     3 trim.mean 0.04682161 -0.0006839994 0.04684004
## 9    250     3    median 0.04958600 -0.0007464279 0.04960519
## 10  1250     3      mean 0.02795388 -0.0004026723 0.02796496
## # ... with 38 more rows
```

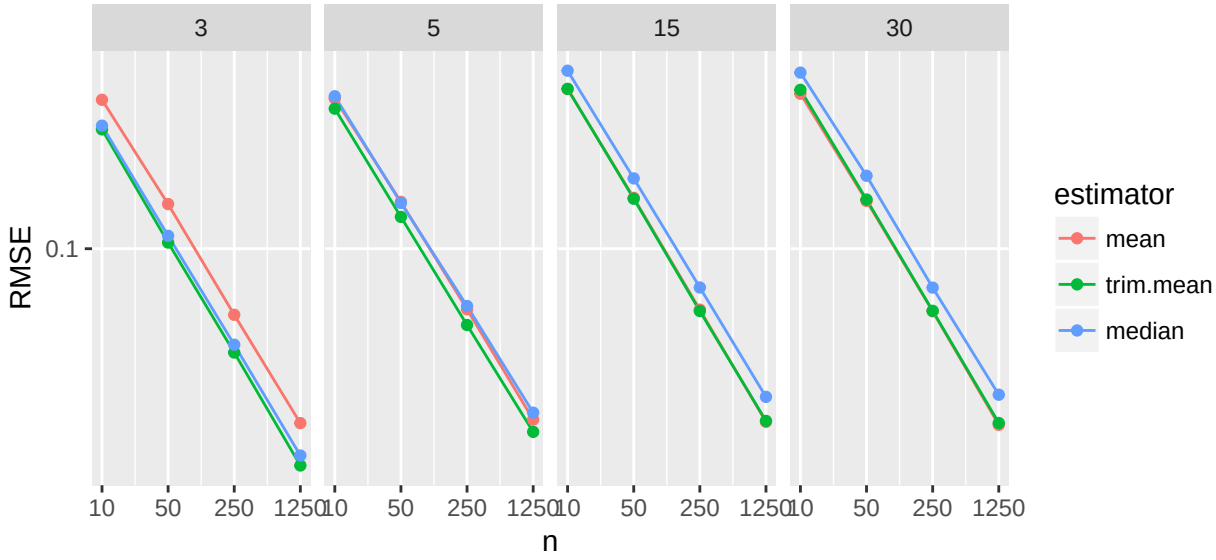
And plot:

```
ggplot( results, aes(x=n, y=RMSE, col=estimator) ) +
            facet_wrap( ~ df, nrow=1 ) +
            geom_line() + geom_point() +
            scale_x_log10( breaks=ns )
```



The above doesn't show differences clearly because all the RMSE goes to zero. It helps to log our outcome, or otherwise rescale. The logging version shows differences are relatively constant given changing sample size.

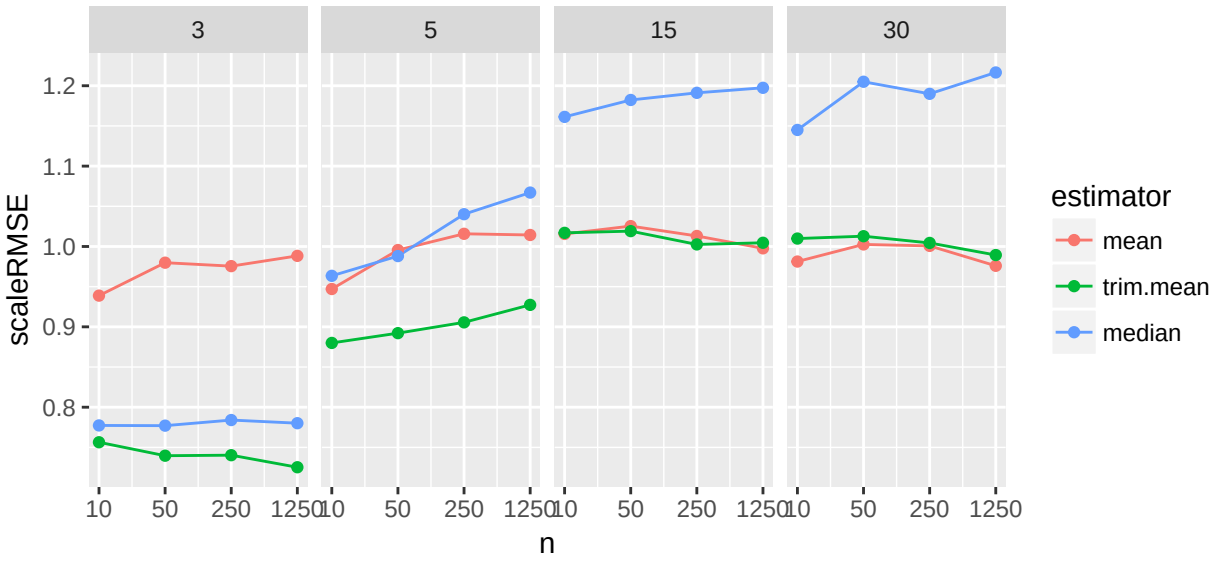
```
ggplot( results, aes(x=n, y=RMSE, col=estimator) ) +
    facet_wrap( ~ df, nrow=1 ) +
    geom_line() + geom_point() +
    scale_x_log10( breaks=ns ) +
    scale_y_log10()
```



Better is to rescale using our knowledge of standard errors. If we scale by the square root of sample size, we should get horizontal lines. We now clearly see the trends.

```
results <- mutate( results, scaleRMSE = RMSE * sqrt(n) )
```

```
ggplot( results, aes(x=n, y=scaleRMSE, col=estimator) ) +
    facet_wrap( ~ df, nrow=1) +
    geom_line() + geom_point() +
    scale_x_log10( breaks=ns )
```



Overall, we see the scaled error of the mean it is stable across the different distributions. The trimmed mean is a real advantage when the degrees of freedom are small. We are cropping outliers that destabilize our estimate which leads to great wins. As the distribution grows more normal, this is no longer an advantage and we get closer to the mean in terms of performance. Here we are penalized slightly bye having dropped 10% of our data, so the standard errors will be slightly larger.

The median is not able to take advantage of the nuances of a data set because it is entirely determined by the middle value. When outliers cause real concern, this cost is minimal. When outliers are not a concern, the median is just worse.

Overall, the trimmed mean seems an excellent choice: in the presence of outliers it is far more stable than the mean, and when there are no outliers the cost of using it is small.

In terms of thinking about designing simulation studies, we see clear visual displays of simulation results can tell very clear stories. Eschew complicated tables with lots of numbers.

Extension: The Bias-variance tradeoff

We can use the above simulation to examine these same estimators when we the median is not the same as the mean. Say we want the mean of a distribution, but have systematic outliers. If we just use the median, or trimmed mean, we might have bias if the outliers tend to be on one side or another. For example, consider the exponential distribution:

```
nums = rexp( 100000 )
mean( nums )
```

```
## [1] 0.99737
```

```
mean( nums, trim=0.1 )
```

```
## [1] 0.8283047
```

```
median( nums )
```

```
## [1] 0.6905491
```

Our trimming, etc., is *biased* if we think of our goal as estimating the mean. But if the trimmed estimators are much more stable, we might still wish to use them. Let's find out.

Let's generate a mixture distribution, just for fun. It will have a nice normal base with some extreme outliers. We will make sure the overall mean, including the outliers, is always 1, however. (So our target, μ is now 1, not 0.)

```
gen.dat = function( n, prob.outlier = 0.05 ) {
    nN = rbinom( 1, n, prob.outlier )
    nrm = rnorm( n - nN, mean=0.5, sd=1 )
    outmean = (1 - (1-prob.outlier)/2) / prob.outlier
    outs = rnorm( nN, mean=outmean, sd=10 )
    c( nrm, outs )
}
```

Let's look at our distribution

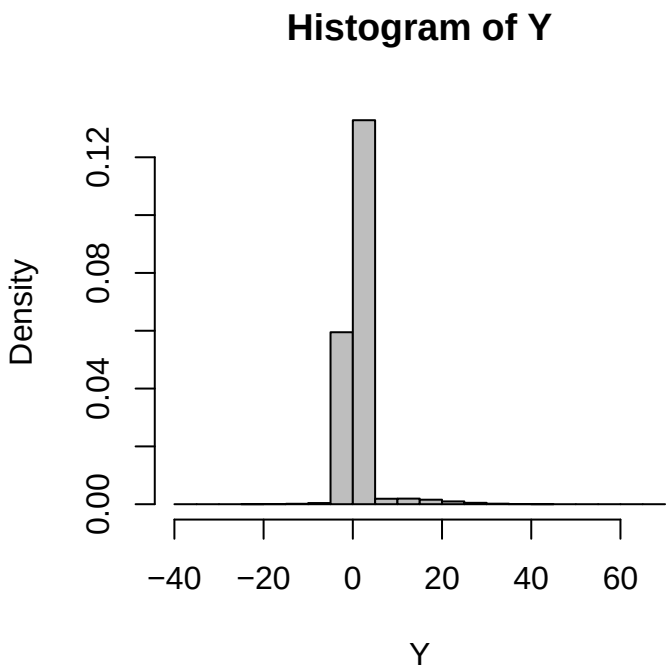
```
Y = gen.dat( 10000000, prob.outlier = 0.05 )
mean( Y )
```

```
## [1] 0.9999024
```

```
sd( Y )
```

```
## [1] 3.270383
```

```
hist( Y, breaks=30, col="grey", prob=TRUE )
```



We steal the code from above, modifying it slightly for our new function and changing our target parameter from 0 to 1:

```
one.run.exp = function( n ) {
    raw.exps <- plyr::rdply( 1000, {
        dt = gen.dat( n=n )
        analyze.data( dt )
    } )
    rs <- plyr::ldply( raw.exps[-1], estimator.quality, mu = 1, .id = "estimator" )
    rs
}
```

```
res = one.run.exp( 100 )
res
```

```
##   estimator      RMSE         bias        SE
## 1      mean 0.3389645  0.003745985 0.3391134
## 2 trim.mean 0.4524438 -0.437268725 0.1162540
## 3    median 0.4730820 -0.454916015 0.1299032
```

And for our experiment we vary the sample size

```
ns = c( 10, 20, 40, 80, 160, 320 )
lvls = tibble( n=ns )
```

```
results <- lvls %>% mutate( results = pmap( lvls, one.run.exp ) ) %>% unnest()
head( results )
```

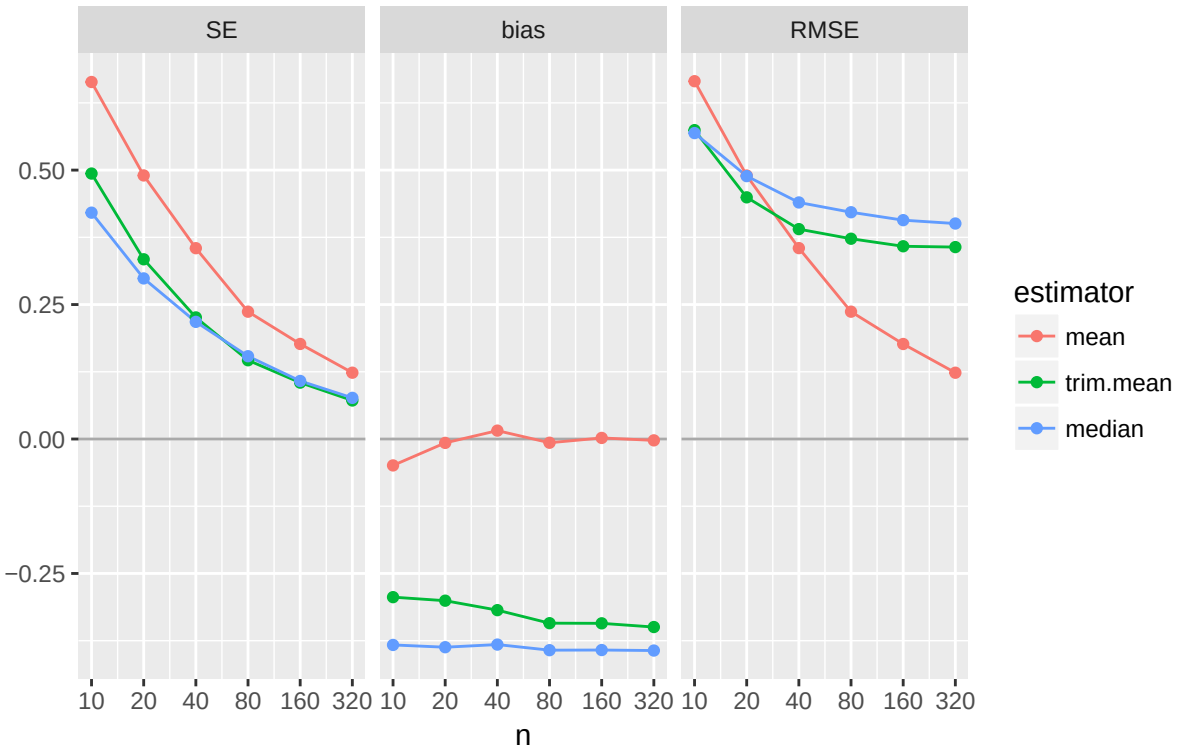
```
## # A tibble: 6 × 5
##       n estimator      RMSE         bias        SE
##   <dbl>    <fctr>     <dbl>        <dbl>     <dbl>
## 1    10      mean 0.6653075 -0.049117897 0.6638239
## 2    10 trim.mean 0.5743417 -0.294033234 0.4936158
## 3    10    median 0.5687996 -0.382882283 0.4208459
## 4    20      mean 0.4898701 -0.007184189 0.4900626
## 5    20 trim.mean 0.4493510 -0.300596230 0.3341704
## 6    20    median 0.4888145 -0.387105453 0.2986271
```

Here we are going to plug multiple outcomes. Often with the simulation study we are interested in different measures of performance. For us, we want to know the standard error, bias, and overall error (RMSE). To plot this we first gather our outcomes to make a long form dataframe of results:

```
res2 = gather( results, RMSE, bias, SE, key="Measure",value="value" )
res2 = mutate( res2, Measure = factor( Measure, levels=c("SE","bias","RMSE" )))
```

And then we plot, making a facet for each outcome of interest:

```
ggplot( res2, aes(x=n, y=value, col=estimator) ) +
    facet_grid( . ~ Measure ) +
    geom_hline( yintercept=0, col="darkgrey" ) +
    geom_line() + geom_point() +
    scale_x_log10( breaks=ns ) +
    labs( y="" )
```



We see how different estimators have different biases and different uncertainties. The bias is negative for our trimmed estimators because we are losing the big outliers above and so getting answers that are too low.

The RMSE captures the trade-off in terms of what estimator gives the lowest overall *error*. For this distribution, the mean wins as the sample size increases because the bias basically stays the same and the SE drops. But for smaller samples the trimming is superior. The median (essentially trimming 50% above and below) is overkill and has too much negative bias.

From a simulation study point of view, notice how we are looking at three different qualities of our estimators. Some people really care about bias, some care about RMSE. By presenting all results we are transparent about how the different estimators operate.

Next steps would be to also examine the associated estimated standard errors for the estimators, seeing if these estimates of estimator uncertainty are good or poor. This leads to investigation of coverage rates and similar.