Software: Science-Wise False Discovery Rate

Nathan (Nat) Goodman

June 1, 2017

Contents

This document describes the software design and design choices of the scripts in my SWFDR GitHub repository. The base script is `swfdr_base.R`, a simple implementation that uses base R capabilities only. Other scripts extend the base implementation by providing solutions to exercises for the reader.

Introduction

The *false discovery rate* (*FDR*) is the probability that a significant p-value indicates a false positive, or equivalently, the proportion of significant p-values that correspond to results without a real effect. FDR estimation is routinely used in genomics for multiple test correction in large scale experiments.

Some time ago I wrote a quick-and-dirty R script that compared FDR estimations methods on simulated data. I decided to adapt this program to explore the notion of science-wise false discovery rate as explained in the companion document Science: Science-Wise False Discovery Rate.

Beyond the interesting science, I thought this would be a good example to get my feet wet in the world of didactic R software aimed at web users. I cleaned up the code, splitting off a simple core and casting the rest as exercises for the reader. I put the code in a GitHub repository and wrote documentation the "R way". Along the way, I expanded my paltry knowledge of GitHib mechanics and learned R documentation tools from scratch.

I hope the software will be a good example for readers who have some R experience but aren't experts. The main script, `swfdr_base.R`, uses base R capabilities only and will run "out of the box" on any reasonably modern R installation. The other scripts extend the base with solutions to some of the exercises for the reader.

It's not a package but rather a collection of functions meant for interactive use, similar to what a non-expert R user is likely to write. It uses a simple, consistent coding style with lots of comments. The software (mostly) conforms to good software engineering practice: the code consists of small functions with well-defined purposes and clear interfaces (i.e., inputs and outputs).

I encourage reader-programmers to download the code, try it out, and modify it to do more. To this end, please see Exercises: Science-Wise False Discovery Rate for ideas on possible extensions. The software is open source, released under the MIT license, which means you can do almost anything you want with it. I'd love to see your improvements if you choose to share them, but you are under no obligation to do so.

Where's the code?

- The `script/` subdirectory of the distribution contains the raw source code.
- The document swfdr_base.code contains the code as a formatted document.
- The HTML version of the present document contains the code in "folded" code blocks as well as links that take you to the right place in the code document.
- The PDF version is less code-friendly. PDF does not support code folding, so I decided to omit long code blocks that would disrupt the flow. Nor does PDF support links to points inside a document, so I can't provide links that take you to the right place in the code document. A reasonable practice, assuming your PDF viewer allows it, is to leave the code document open, display the table of contents in the sidebar, and navigate to code sections as needed.

Scenario

The program simulates a large number of problem instances representing published results, some of which are true and some false. The instances are very simple: we generate two groups of random numbers and use the t-test to assess the difference between their means. One group (the control group or simply *group0*) comes from a standard normal distribution with mean=0. The other group (the treatment group or simply *group1*) is a little more involved:

- for *true* instances, we take numbers from a standard normal distribution with mean d ($d > 0$);
- for *false* instances, we use the same distribution as group0.

The parameter d is the effect size, aka *Cohen's d*.

We use the t-test to compare the means of the groups and produce a p-value assessing whether both groups comes from the same distribution.

We do this thousands of times (drawing different random numbers each time, of course), collect the resulting p-values, and compute the FDR. We repeat the procedure for a range of assumptions to determine the conditions under which most published results are wrong.

For *true* instances, we expect the difference in means to be approximately d and for *false* ones to be approximately 0, but due to the vagaries of random sampling, this may not be so. If the actual difference in means is far from the expected value, the t-test may get it wrong, declaring a *false* instance to be positive and a *true* one to be negative. The goal is to see how often we get the wrong answer across a range of assumptions.

Nomenclature

To reduce confusion, I will be obsessively consistent in my terminology.

- An *instance* is a single run of the simulation procedure.

- The terms *positive* and *negative* refer to the results of the t-test. A *positive instance* is one for which the t-test reports a significant p-value; a *negative instance* is the opposite. Obviously the distinction between positive and negative rests on the chosen significance level.
- *true* and *false* refer to the correct answers. A *true instance* is one where the treatment group (group1) is drawn from a distribution with $mean = d$ ($d > 0$). A *false instance* is the opposite: an instance drawn from a distribution with $mean = 0$.
- *empirical* refers to results calculated from the simulated data.

Programming Style

As you examine the software, you'll no doubt notice that my R code looks different from other examples out there and eschews many recommendations in well-respected R style guides by Hadley Wickham and Google. One obvious difference is that my code is scrunched with less whitespace than others. In addition, I use `=` instead of `<-`, prefer single-quotes over double, end statements with semi-colons, start full line comments with ##, and use `.` as a word separator in variable names and `_` in function names. I have good reasons for some of these choices, but mostly it's just habit.

Programming style is a matter of taste (unless your boss foists a style on you). My style works for me. As you gain experience, I encourage you to find a style that works for you.

Software (swfdr_base.R)

`run`

The top-level function is `run`. The code block below shows how to invoke `run` with default parameters assuming your working directory is the root of the repository.

```
source("script/swfdr_base.R");
run();
```

`run` has 4 steps.

```
run=function(...) {
  init(...);                        # process parameters & other initialization
  doit();                           # do it!
  saveit(save.rdata,save.txt);      # optionally save parameters and results
  fini();                           # final cleanup if any
}
```

- `init` sets everything up
- `doit` (pronounced "do it!") does all the work
- `saveit` (pronounced "save it!") optionally saves the parameters, figures, and results
- `fini` does final cleanup if any; there's nothing to do in this example, but I include it for stylistic consistency.

Global Variables

A key consideration is this sort of program is how to pass parameters and results from one step to the next. I decided to use the very simple scheme of passing everything in global variables. This runs counter to conventional wisdom in the programming world but seemed okay provided it was well-documented. It's also conducive for interactive use.

The table below lists the global variables. The first group are simulation parameters, next are parameters that control program operation, and last are results.

variable	meaning	default or source
prop.true	fraction of cases where there is a real effect	`seq(.1,.9,by=.2)`
m	number of iterations	`1e4`
n	sample size	`16`
d	standardized effect size (aka *Cohen's d*)	`c(.25,.50,.75,1,2)`
pwr	power. if set, program adjusts *d* to achieve power	`NA`
sig.level	significance level for power calculations when *pwr* is set	`0.05`
pval.plot	p-values for which we plot results	`c(.001,.01,.03,.05,.1)`
scriptname	used to set output directories and in error messages	`'swfdr_base'`
datadir	data directory relative to distribution root	`'data/swfdr_base'`
figdir	figure directory relative to distribution root	`'figure/swfdr_base'`
save	save parameters, results, and plots; sets *save.rdata* and *save.plot*, not *save.txt*	`FALSE`
save.rdata	save parameters and results in RData format	`FALSE` (set by *save*)
save.txt	save results in txt format. **CAUTION: big and slow**	`FALSE` (not set by *save*)
save.plot	save plots	`FALSE` (set by *save*)
clean	remove contents of data and figure directories; sets *clean.data* and *clean.fig*	`FALSE`
clean.data	remove contents of data directory	`FALSE` (set by *clean*)
clean.fig	remove contents of figure directory	`FALSE` (set by *clean*)
cases	parameter grid	created by `init`
sim	simulation results	created by `dosim`
interp	interpolation results	created by `dointerp`

Initialize and Finalize

`init` sets the parameters. (Code not shown).

`init`'s argument list defines all parameters and provides default values. To change a parameter, simply call `run` (or `init` directly) with the parameter and new value, e.g., `run(n=32)` changes the sample size *n* from its default (16) to the new value (32). `init` creates a parameter grid, called *cases*, containing all combinations of parameters. The code section at the bottom of `init` copies all parameters and its result, cases, into global variables.

For the default parameters, the parameter grid expands to 25 cases (5 values of *prop.true* $\times$ 5 values of *d*; all other parameters have single-valued defaults). We do 10,000 simulations for each case for a total of 250,000 simulations. This takes about 3 minutes on my small Linux server.

`fini` does final cleanup if any. There's nothing to do in this example, but I include it for stylistic consistency. (Code not shown).

`doit`

`doit` has 3 steps.

```
doit=function() {
  dosim();                          # do simulation
  dointerp();                          # interpolate at fixed pvals
  doplot(save.plot);                   # plot results and optionally save plots
}
```

- `dosim` runs the simulation
- `dointerp` interpolates relevant columns of the simulation results at fixed p-values so we can plot results for p-values of interest
- `doplot` plots the results and optionally save the plots.

Simulation Functions

`dosim` considers cases one by one, calling `sim_one` to do the heavy lifting, and stores the result in a global data frame, called *sim*.

```
dosim=function() {
  ## call sim_one on each case and combine results into data frame
  ## the complicated one-liner below is a good idiom for doing this
  sim=do.call(rbind,apply(cases,1,
    function(case) do.call("sim_one",as.list(case)[c('prop.true','m','n','d')])));
  sim<<-sim;
  invisible(sim);
}
```

`sim_one` does the actual simulations. (Code not shown).

The program draws m random samples of size n for each of the two groups, storing the numbers for each group in a $m \times n$ matrix whose rows are instances and columns are samples. The samples for group0 are from a standard normal distribution with mean=0, i.e. `rnorm(n,mean=0)`. For group1, the code computes the requisite numbers of true and false instances (*num.true* and *num.false*, resp.), draws *num.true* samples from a standard normal distribution with mean=d, i.e. `rnorm(n,mean=d)` and *num.false* samples from the same distribution as group0.

The program runs the t-test on each of the m pairs of samples, generating m p-values, and calculates theoretical and empirical FDRs for each p-value. It also calculates and stores basic statistics for each pair of samples, e.g., the actual means and differences, should this be of interest later.

`sim_one` calls functions `fdr_theo` and `fdr_empi` to do the FDR calculations.

FDR Calculations

Recall that FDR is the number of false positives divided by the total number of positives, and of course, the total number of positives is the number of false positives plus the number of true ones. In pseudo-math: $FDR = num.fp/(num.fp + num.tp)$.

The theoretical FDR uses textbook formulas to estimate *num.fp* and *num.tp* for each p-value *pval*.

```
fdr_theo=function(pval,num.true,num.false,n=16,d=1) {
  pwr=sapply(pval,function(p) power.t.test(n,delta=d,sd=1,sig.level=p)$power);
  num.tp=pwr*num.true;
  num.fp=pval*num.false;
  num.fp/(num.tp+num.fp);
}
```

- *num.fp* = *pval* times the number of false instances = *pval* $\times$ (1 $-$ *prop.true*) $\times$ m
- *num.tp* = power (given *pval*) times the number of true instances = *power* $\times$ *prop.true* $\times$ m

The program calculates power using R's power.t.test function: `power.t.test(n=n,delta=d,sd=1,sig.level=p)$power`.

The empirical FDR is conceptually simpler.

```
fdr_empi=function(pval,d.true) {
  order=order(pval);
  pval=pval[order];
  d.true=d.true[order];
  m=length(pval);
  neg=cumsum(ifelse(d.true,0,1));
  fdr=neg/(1:m);        # because pval is sorted, index is number of entries with smaller pval
                        # equals number of entries that would be accepted at this pval
  ## deal with ties if necessary
  ## if (anyDuplicated(pval)) {
  ##   ## split fdr by pval, then set fdr to max fdr of group.
  ##   ## CAUTION: have to deal with approximate equality of pvals
  ##   digits=ceiling(-log10(min(pval)));  # number of significant digits in smallest pval
  ##   pval.approx=signif(pval,digits);
  ##   fdr.by.pval=split(fdr,pval.approx);
  ##   max.by.pval=sapply(fdr.by.pval,function(g) if (all(is.na(g))) NA else max(g,na.rm=T));
  ##   fdr=sapply(pval.approx,function(p) max.by.pval[as.character(p)]);
  ## }
  unorder=order(order); # restore original order
  fdr[unorder];
}
```

For each p-value *pval*, the code counts the number of false positives in instances with p-values $\leq$ *pval* and divides by the number of all instances with such p-values. In the unlikely, but possible, event that multiple instances have the same p-value, this calculation would probably give different answers for each duplicate. The code handles this case by setting the empirical FDR to the maximum value for all instances with the given p-value.

Interpolation Functions

The structure of `dointerp` is similar to `dosim`. It calls `interp_one` on each case to interpolate the theoretical and empirical FDRs at desired p-values using R's `approxfun` function. `dointerp` stores the result in a global data frame, called *interp*.

```
dointerp=function() {
  ## call interp_one on each case and combine results into data frame
  ## the complicated one-liner is a good idiom for doing this
  interp=do.call(rbind,apply(cases,1,
    function(case) do.call("interp_one",as.list(case)[c('prop.true','m','n','d')])));
  interp<<-interp;
  invisible(interp);
}
interp_one=function(prop.true,m,n,d) {
  sim1=sim[(sim$prop.true==prop.true&sim$m==m&sim$n==n &sim$d==d),];
  fun_theo=with(sim1,approxfun(pval,fdr.theo,rule=2));
  fdr.theo=fun_theo(pval.plot);
  fun_empi=with(sim1,approxfun(pval,fdr.empi,rule=2));
  fdr.empi=fun_empi(pval.plot);
  interp=data.frame(prop.true,m,n,d,pval=pval.plot,fdr.theo,fdr.empi);
  invisible(interp);
}
```

Plot Functions

These functions operate on the `sim` and `interp` data frames produced by `dosim` and `dointerp` respectively.

`doplot` is the main plot function. It calls separate functions for each of the four kinds of plot.

```
doplot=function(save.plot=F) {
    plot_byprop(save.plot);             # plot fdr by prop.true
    plot_byd(save.plot);                # plot fdr by d
    plot_vsprop(save.plot);             # plot fdr for fixed pvals vs prop.true
    plot_vsd(save.plot);                # plot fdr for fixed pvals vs d
  }
```

- `plot_byprop` plots FDR by *prop.true* for one value of *d*. (Code not shown).
- `plot_byd` plots FDR by *d* for one value of *prop.true*. (Code not shown).
- `plot_vsprop` plots FDR vs *prop.true* for one value of *d* at fixed p-values. (Code not shown).
- `plot_vsd` plots FDR vs *d* for one value of *prop.true* at fixed p-values. (Code not shown).

The four plot functions are similar. Each function plots a different slice of theoretical and empirical FDR as a function of three variables: $FDR = f(prop.true, d, pval)$. The plot functions differ in how they reduce four dimensions (FDR and the three variables) to something that can be plotted in two dimensions.

Each function starts by fixing one variable to a single value. Next, the function splits the data into groups based on a second variable. Finally, it plots each group vs. the remaining variable, using different line types (solid vs dashed) to distinguish theoretical and empirical FDR.

Much of the code in each function deals with plotting mechanics: setting up the axes, title, legend, and grid lines, and optionally saving the plots.

`plot_byprop` and `plot_byd` operate on *sim*; `plot_vsprop` and `plot_vsd` operate on *interp*.

Save and Load

`saveit` saves all parameters and results (actually, all global variables) in RData and optionally tab-delimited text formats. (Code not shown).

The companion function, `loadit`, loads the saved data into the R session. (Code not shown).

Utility Functions

There are two utility (helper) functions. (Code not shown).

- `power_grid` adds power calculations to the *cases* data frame
- `colramp` sets up the colors for the plot functions.

Conclusion or Wrap-up TBD