How to make scripts work

Matthew Suderman

Lecturer in Epigenetic Epidemiology

MRC Integrative Epidemiology Unit

University of BRISTOL

"My script doesn't work"

Two main problems

1. script fails
2. script seems to run forever

Example: dimension reduction

We want to reduce one set of variables to a smaller set that captures most of the same variation.

To make this interesting, we'll also require that certain effects are omitted from the output variables.

Example: dimension reduction

We want to reduce one set of variables to a smaller set that captures most of the same variation.

To make this interesting, we'll also require that certain effects are omitted from the output variables.

Inputs

```
> round(variables[1:5,1:5],3)
         var1   var2   var3   var4   var5
sample1  0.181 -0.983 -0.081  1.301 -1.274
sample2  0.114 -0.303 -0.289 -0.883  0.821
sample3 -0.531 -1.014  0.091 -0.392 -1.156
sample4 -1.018  2.266  0.787 -0.227  0.585
sample5 -0.090  0.041  0.431  1.446  0.366
```

```
> round(effects[1:5,],3)
        effect1 effect2 effect3 effect4
sample1   0.773   0.072  -2.312  -1.064
sample2   0.077  -0.151   0.278  -1.066
sample3   0.103  -0.095  -0.471   0.633
sample4   0.058  -0.028  -0.017  -0.251
```

Outputs

A matrix like variables with fewer columns that captures most of the variance in variables but none of effects.

Main function definition

```
## Captures variance with fewer variables
## Parameters:
## - variables: matrix of variables (columns=variables, rows=observations)
## - effects: matrix of effects (columns=effects, rows=observations)
## - num: number of desired output variables
## Output:
## A matrix of `num` variables that capture variation in `variables`
## but none of the variation in `effects`.
##
reduce.dimensions <- function(variables, effects, num) {
   variables <- scale(variables)                    ## standardize variables (mean=0, sd=1)
   variables <- remove.effects(variables, effects)      ## remove effects from variables
   reduced.variables <- compute.pcs(variables, num)      ## reduce variables to `num` variables
   reduced.variables
}
```

Assuming we have a set of variables and effects, we apply reduced.dimensions as follows:

```
new.vars <- reduced.dimensions(variables,effects,5)
```

scale() is already available in R.

Defining remove.effects() and compute.pcs()

```
## Removes effects from variables
## Parameters:
## - variables: matrix of variables (columns=variables, rows=observations)
## - effects: matrix of effects (columns=effects, rows=observations)
## Output:
## The residuals of fitting the model "variables ~ effects".
##
remove.effects <- function(variables, effects) {
    fit <- lm.fit(x=effects, y=variables)
    residuals(fit)
}

## Obtain the top principal components
## Parameters:
## - variables: matrix of variables (columns=variables, rows=observations)
## - num: number of desired output variables
## Output:
## The top `num` principal components of `variables`.
##
compute.pcs <- function(variables, num) {
    fit <- prcomp(variables)                    ## perform PCA
    pcs <- fit$x                                ## extract PCs
    pcs <- pcs[,1:num,drop=F]                   ## select top `num` PCs
```

Reducing dimensions

Simulate some data

```
> n <- 50                ## 50 observations/samples
> n.vars <- 20             ## 20 variables
> n.effs <- 5             ## 5 effects to remove

> variables <- matrix(rnorm(n.vars*n),nrow=n)
> effects <- matrix(rnorm(n.effs*n), nrow=n)
```

```
> dim(variables)
[1] 50 20
> dim(effects)
[1] 50  5
> dim(new.vars)
[1] 50  5
```

Reduce dimensions

```
> new.vars <- reduce.dimensions(variables,effe
```

Reducing dimensions

Simulate some data

```
> n <- 50                ## 50 observations/samples
> n.vars <- 20           ## 20 variables
> n.effs <- 5            ## 5 effects to remove

> variables <- matrix(rnorm(n.vars*n),nrow=n)
> effects <- matrix(rnorm(n.effs*n), nrow=n)
```

```
> dim(variables)
[1] 50 20
> dim(effects)
[1] 50  5
> dim(new.vars)
[1] 50  5
```

Reduce dimensions

```
> new.vars <- reduce.dimensions(variables,effe
```

Check outputs (i.e. show correlations between variables and effects)

```
> quantile(cor(variables, effects))
    0%    25%    50%    75%   100%
-0.257 -0.116 -0.005  0.096  0.356

> quantile(cor(new.vars, effects))
       0%       25%       50%       75%      100%
-1.18e-02 -1.25e-03 -1.44e-05  1.02e-03  9.73e-

> quantile(cor(new.vars, variables))
    0%    25%     50%    75%   100%
-0.647 -0.194  0.0381  0.251  0.547
```

Encountering an error

Suppose that one of our initial variables had no variance. We simulate this:

```
> variables[,4] <- 0
```

Reducing dimensions generates an error.

```
> new.vars <- reduce.dimensions(variables,effec
Error in lm.fit(x = effects, y = variables) :
  NA/NaN/Inf in 'y'
```

Encountering an error

Suppose that one of our initial variables had no variance. We simulate this:

```
> variables[,4] <- 0
```

Reducing dimensions generates an error.

```
> new.vars <- reduce.dimensions(variables,effec
Error in lm.fit(x = effects, y = variables) :
  NA/NaN/Inf in 'y'
```

Strange, lm.fit() is complaining about missing values, but neither variables nor effects had missing values when we started.

Because our code is short, it is easy to figure out where the error is being generated. For longer more complex code, the traceback function can help with this:

```
> traceback()
3: lm.fit(x = effects, y = variables) at example.r
2: remove.effects(variables, effects) at example
1: reduce.dimensions(variables, effects, 5)
```

i.e. reduce.dimsions() called remove.effects() which called lm.fit(). That's easy to find.

Identifying the problem using debug()

We now tell R that we would like to 'step' through the code in reduce.dimensions() to determine where missing values are being introduced.

```
> debug(reduce.dimensions)
> new.vars <- reduce.dimensions(variables,effe
```

R now stops at the beginning of the reduce.dimensions function to allow us to say what to do next.

```
debugging in: reduce.dimensions(variables, effe
debug at /tmp/example.r@46#1: {
    variables <- scale(variables)
    variables <- remove.effects(variables, effects)
    reduced.variables <- compute.pcs(variables, r
    reduced.variables
}
```

Identifying the problem using debug()

We now tell R that we would like to 'step' through the code in reduce.dimensions() to determine where missing values are being introduced.

```
> debug(reduce.dimensions)
> new.vars <- reduce.dimensions(variables,effe
```

R now stops at the beginning of the reduce.dimensions function to allow us to say what to do next.

```
debugging in: reduce.dimensions(variables, effe
debug at /tmp/example.r@46#1: {
    variables <- scale(variables)
    variables <- remove.effects(variables, effects)
    reduced.variables <- compute.pcs(variables, r
    reduced.variables
}
```

We check if there are any missing values in variables. There are none.

```
Browse[2]> any(is.na(variables))
[1] FALSE
```

We then type 'n' to run the next line in the function.

```
Browse[2]> n
debug at /tmp/example.r@46#2:
  variables <- scale(variables)
```

Variables have now been scaled. We check again for missing values.

```
Browse[2]> any(is.na(variables))
[1] TRUE
```

debug() continued

This time there are missing values. We check where they are in variables.

```
Browse[2]> which(is.na(variables),arr.ind=T)
     row col
[1,]   1   4
[2,]   2   4
[3,]   3   4
[4,]   4   4
 ...
```

We can see that they are in column 4, this has something to do this scaling a constant variable ...

... we can solve this by removing constant variables from the dataset.

debug() continued

This time there are missing values. We check where they are in variables.

```
Browse[2]> which(is.na(variables),arr.ind=T)
     row col
[1,]   1   4
[2,]   2   4
[3,]   3   4
[4,]   4   4
 ...
```

We can see that they are in column 4, this has something to do this scaling a constant variable ...

... we can solve this by removing constant variables from the dataset.

We've discovered the problem, so we quit debugging.

```
Browse[2]> Q
>
```

This returns us to the usual R prompt.

We also tell R that we don't want to debug reduce.dimensions() anymore.

```
undebug(reduce.dimensions)
```

Removing constant variables

Below is a function for identifying and removing variables that do not vary.

```
remove.constants <- function(variables) {
   ss <- rep(NA,ncol(variables))                    ## `ss` will hold the variances
                                          ##        of each variable
   for (i in 1:ncol(variables))                      ## for each variable i
      ss[i] <- var(variables[,i])                   ##   ss[i] = variance of variable i

   is.constant <- ss < 2e-16                         ## is.constant == TRUE for all variables
                                          ##        with variance < 2x10^-16
   if (any(is.constant)) {                          ## if any variable has low variance
      warning("Omitting variables with zero variance: ",  ##   issue a warning that they will be removed
            sum(is.constant))                       ##
      variables <- variables[,!is.constant,drop=F]       ##   remove those variables
   }
   variables
}
```

It works!

Apply the new function to variables. It should remove variable 4.

```
> new.vars <- remove.constants(variables)
Warning message:
In remove.constants(variables) :
  Omitting variables with zero variance: 1
```

One variable is indeed missing.

```
> dim(new.vars)
[1] 50 19
```

Variable 4 was removed.

```
> identical(new.vars[,4], variables[,5])
[1] TRUE
```

It works!

Apply the new function to variables. It should remove variable 4.

```
> new.vars <- remove.constants(variables)
Warning message:
In remove.constants(variables) :
  Omitting variables with zero variance: 1
```

One variable is indeed missing.

```
> dim(new.vars)
[1] 50 19
```

Variable 4 was removed.

```
> identical(new.vars[,4], variables[,5])
[1] TRUE
```

Adding the remove.constants() fixes the problem in reduce.dimensions().

```
reduce.dimensions<-function(variables, effects,
   variables <- remove.constants(variables) ###
   variables <- scale(variables)
   variables <- remove.effects(variables, effects)
   reduced.variables <- compute.pcs(variables, r
   reduced.variables
}

> new.vars <- reduce.dimensions(variables, effec
```

Identifying problems using browser()

browser() is similar to debug() except that it is inserted into the code where you'd like to stepping through the code.

To demonstrate browser(), we will add a missing values to variables.

```
> variables[3,4] <- NA
```

Applying reduce.dimensions() to variables generates a new error.

```
> new.vars <- reduce.dimensions(variables, effects, 5)
## Error in if (any(is.constant)) { : missing value where TRUE/FALSE needed
```

traceback() indicates the error is in our new remove.constants() function.

```
> traceback()
2: remove.constants(variables) at example.r@55#2
1: reduce.dimensions(variables, effects, 5)
```

browser() continued

We insert browser() immediately before the if-statement where the error is generated.

```
remove.constants <- function(variables) {
   ss <- rep(NA,ncol(variables))
   for (i in 1:ncol(variables))
      ss[i] <- var(variables[,i])
   browser()                          ######
   is.constant <- ss < 2e-16
   if (any(is.constant)) {               ## where the error was generated
      warning("Omitting variables with zero variance: ", sum(is.constant))
      variables <- variables[,!is.constant,drop=F]
   }
   variables
}
```

browser() continued

Running reduce.dimensions() again presents us with a debug prompt immediately before the if-statement.

```
> new.vars <- reduce.dimensions(variables, effects, 5)

Called from: remove.constants(variables)
Browse[1]> debug at /tmp/example.r@52#6: is.constant <- ss < 2e-16
Browse[2]> n
debug at /tmp/example.r@52#7: if (any(is.constant)) {
    warning("Omitting variables with zero variance: ", sum(is.constant))
    variables <- variables[, !is.constant, drop = F]
}
```

The error was about a missing value in is.constant. We check which one.

```
Browse[2]> which(is.na(is.constant))
[1] 4

Browse[2]> var(variables[,4])
[1] NA
```

browser() continued

var() returns a missing value whenever *any* input value is missing. We can override this by setting na.rm=TRUE (i.e. remove missing values before calculating variance).

```
Browse[2]> var(variables[,4],na.rm=T)
[1] 0.8257385

Browse[2]> Q
```

We now remove browser() and add na.rm=T to our function.

```
remove.constants <- function(variables) {
   ss <- rep(NA,ncol(variables))
   for (i in 1:ncol(variables))
      ss[i] <- var(variables[,i], na.rm=T) ######
   is.constant <- ss < 2e-16
   if (any(is.constant)) {
      warning("Omitting variables with zero variance: ", sum(is.constant))
      variables <- variables[,!is.constant,drop=F]
   }
   variables
}
```

Catching errors with stopifnot()

Sometimes our code should just stop when an error is encountered, e.g.

```
> new.vars <- reduce.dimensions(variables, effects, 55)
Error in pcs[, 1:num, drop = F] : subscript out of bounds

> traceback()
2: compute.pcs(variables, num) at example.r@51#5
1: reduce.dimensions(variables, effects, 55)
```

The function stopifnot() tells R to stop if a certain condition is not met.

```
reduce.dimensions <- function(variables, effects, num) {
   variables <- remove.constants(variables)
   stopifnot(num <= ncol(variables))
   ...
}
```

This generates a more meaningful error message.

```
> new.vars <- reduce.dimensions(variables, effects, 55)
Error in reduce.dimensions(variables, effects, 55) :
```

Catching errors with stop()

Alternatively, you could use the stop() function.

```
reduce.dimensions <- function(variables, effects, num) {
    variables <- remove.constants(variables)
    if (num > ncol(variables))
      stop("num = ", num, " must be at most ", ncol(variables))
    ...
}
```

```
> new.vars <- reduce.dimensions(variables, effects, 55)
Error in reduce.dimensions(variables, effects, 55) :
  num = 55 must be at most 20
```

Is this a better error message?

system.time() -- How long did that take anyway?

We simulate a larger dataset that will require more computation.

```
> n <- 1000              ## 1000 observations/samples
> n.vars <- 5e4           ## 50K variables
> n.effs <- 50           ## 50 effects to remove
> variables <- matrix(rnorm(n.vars*n),nrow=n)
> effects <- matrix(rnorm(n.effs*n), nrow=n)
```

system.time() -- How long did that take anyway?

We simulate a larger dataset that will require more computation.

```
> n <- 1000              ## 1000 observations/samples
> n.vars <- 5e4           ## 50K variables
> n.effs <- 50           ## 50 effects to remove
> variables <- matrix(rnorm(n.vars*n),nrow=n)
> effects <- matrix(rnorm(n.effs*n), nrow=n)
```

We run the command as previously but within system.time(...).

```
> system.time(new.vars <- reduce.dimensions(variables, effects, 5))
   user  system elapsed
 48.784   5.243  54.091
```

system.time() -- How long did that take anyway?

We simulate a larger dataset that will require more computation.

```
> n <- 1000              ## 1000 observations/samples
> n.vars <- 5e4           ## 50K variables
> n.effs <- 50           ## 50 effects to remove
> variables <- matrix(rnorm(n.vars*n),nrow=n)
> effects <- matrix(rnorm(n.effs*n), nrow=n)
```

We run the command as previously but within system.time(...).

```
> system.time(new.vars <- reduce.dimensions(variables, effects, 5))
   user  system elapsed
 48.784   5.243  54.091
```

- 48.784 seconds spent on the command
- 5.243 seconds spent on unrelated system activities
- 54.091 seconds total from start to finish ('wall time')

Rprof() -- Oh, that's what took so long!

Start the profiler

```
> Rprof()
```

Run the command as previously and obtain the profile summary.

```
> new.vars <- reduce.dimensions(variables, eff
> profile <- summaryRprof()
```

We can see how long each function was 'active'.

```
> profile$by.total
                  total.time total.pct self.time self.pct
reduce.dimensions      52.26    100.00      0.00      0.0
prcomp.default         39.98     76.50      0.34      0.65
compute.pcs            39.98     76.50      0.00      0.00
prcomp                 39.98     76.50      0.00      0.00
```

Rprof() -- Oh, that's what took so long!

Start the profiler

```
> Rprof()
```

Run the command as previously and obtain the profile summary.

```
> new.vars <- reduce.dimensions(variables, eff
> profile <- summaryRprof()
```

We can see how long each function was 'active'.

```
> profile$by.total
                total.time total.pct self.time self.pct
reduce.dimensions      52.26    100.00      0.00      0.0
prcomp.default         39.98     76.50      0.34      0.65
compute.pcs            39.98     76.50      0.00      0.00
prcomp                 39.98     76.50      0.00      0.00
```

We can also see which function was most active minus activity of functions called by the function. This is probably more useful for reducing running time.

```
> profile$by.self
                  self.time self.pct total.time total.pct
La.svd                32.32    61.84      33.00     63.15
lm.fit                 5.52    10.56       5.52     10.56
%*%                    3.56     6.81       3.56      6.81
aperm.default          3.34     6.39       3.34      6.39
t.default              0.88     1.68       0.88      1.68
array                  0.86     1.65       0.86      1.65
stopifnot              0.64     1.22       0.72      1.38
...
remove.constants       0.08     0.15       1.96      3.75
 ...
```

Finally, turn profiling off.

```
> Rprof(NULL)
```

Summary

- traceback() -- where was the error generated

Summary

- traceback() -- where was the error generated
- debug() and browser() -- what caused the error

Summary

- traceback() -- where was the error generated
- debug() and browser() -- what caused the error
- warning() -- something might not be quite right

Summary

- traceback() -- where was the error generated
- debug() and browser() -- what caused the error
- warning() -- something might not be quite right
- stopifnot() and stop() -- something is definitely not right

Summary

- traceback() -- where was the error generated
- debug() and browser() -- what caused the error
- warning() -- something might not be quite right
- stopifnot() and stop() -- something is definitely not right
- system.time() -- how long did that take?

Summary

- traceback() -- where was the error generated
- debug() and browser() -- what caused the error
- warning() -- something might not be quite right
- stopifnot() and stop() -- something is definitely not right
- system.time() -- how long did that take?
- Rprof() -- what was it doing all that time?

More information

- https://rstats.wtf/debugging-r-code.html
- https://bookdown.org/rdpeng/rprogdatascience/debugging.html
- https://rstudio-education.github.io/hopr/debug.html
- https://bookdown.org/rdpeng/rprogdatascience/profiling-r-code.html