

Data Programming with R

Isabella Gollini

Lecture 10 - Debugging

Debugging

- Simple tips
- R's debugging tools
 - `browser`
- A simple example
- Other debugging tools
- Setting breakpoints
- Extended example

What is debugging?

- Debugging is the process of checking your code and finding and fixing mistakes.
- You will already have encountered many occasions where your R code does not work: you typed in something incorrect and need to change it. R will usually (try to) tell you what you have done wrong.
- Worse, sometimes your code works but does not give the answers you expected. You need to locate where the problem is in the code and fix it.
- You will often spend more time debugging code than writing it!
- In this lecture we will go through the different tools R & Rstudio have for checking and finding mistakes in R code.

Tips for debugging

- Put comments into your code as you go.
- Start small. Try some very simple test cases and avoid working with large data objects.
- Debug in a modular, top-down manner. Try to write a main function which isn't very long which calls other functions as it goes through. It's easier to debug and check small individual functions than one long huge one.
- Use anti-debugging. The `stop()` and `stopifnot()` functions will stop a function if it has a dodgy value in it:

```
x <- 5
if(length(x) != 3) stop("x is supposed to be of length 3!")

## Error in eval(expr, envir, enclos): x is supposed to be of length 3!
stopifnot(x < 0)

## Error in eval(expr, envir, enclos): x < 0 is not TRUE
```

Using debugging tools

- The simplest way of debugging is to simply insert `print` commands into your code to check that it is returning the correct values.
- This is an incredibly slow way to debug and can prove distracting as the `print` statements can contain bugs too.

- Instead R contains a very useful function, **browser**, which will allow you to explore the internals of a function whilst you are running it.

Using browser

- The **browser** function, when inserted into another function, allows you to explore the workspace inside the function as it is being run.
- When **browser** has been called, the R prompt changes from `>` to `Browse[d]>` where `d` is how many environments we are nested in.
- When in **browser** mode there are number of special commands:
 - **n** (for next): advances the program by one line
 - **c** (for continue): will execute the rest of the function and stop again if it hits the **browser** command (which might not happen). You can use Enter for this too.
 - **where**: will give you the stack trace (i.e. the functions that were called beforehand)
 - **Q**: will quit **browser** and go back to the location above where **browser** was called (usually back to the R command prompt).

Some notes about browser

- Whilst in **browser** mode you can execute any R command you like. It is usually desirable to explore the objects in the current workspace with commands like **print**, **plot**, **head**, etc.
- Be careful if you have objects named **n** or **c** as the **browser** commands will take priority over them. If you have an object named **n** use **print(n)** to see it.
- You can have multiple calls to **browser** in different functions. The prompt will tell you how ‘nested’ your current **browser** call is.
- In the next few slides we go through a simple example.

A simple debugging example



- Debugging **neglikfun**

A simple wrong example

- Recall the likelihood example of lecture 6:

```
neglikfun <- function(parameters) {
  beta0 <- parameters[1]
  beta1 <- parameters[2]
  daty0 <- y == 0
  daty1 <- y == 1
  loglikelihood <- sum(log(1 / (1 + exp(-beta0 - beta1*x[daty1])))) +
    sum(log( - (1 / (1 + exp(-beta0 - beta1*x[daty0])))))
  return(-loglikelihood)
}
neglikfun(c(1,-0.1))
```

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

- We’ve made a mistake somewhere, but where?

Debugging the likelihood

- If we insert `browser()` just after the function definition and then run it we end up with:

```
neglikfun <- function(parameters) {  
  browser()  
  beta0 <- parameters[1]  
  beta1 <- parameters[2]  
  daty0 <- y == 0  
  daty1 <- y == 1  
  loglikelihood <- sum(log(1 / (1 + exp(-beta0 - beta1*x[daty1])))) +  
    sum(log( - (1 / (1 + exp(-beta0 - beta1*x[daty0])))))  
  return(-loglikelihood)  
}  
neglikfun(c(1,-0.1))
```

Debugging the likelihood

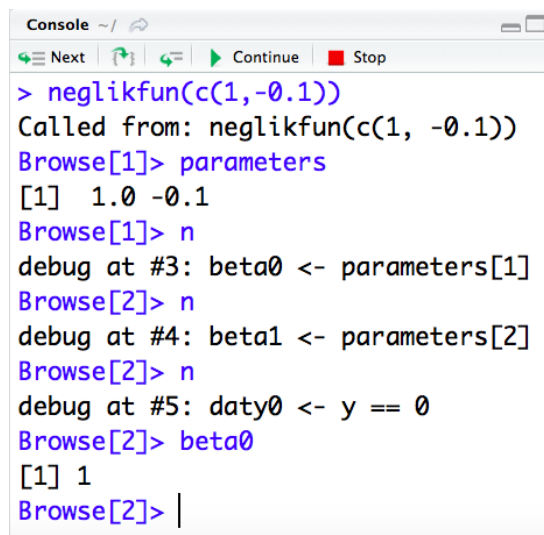
- We can now explore the R workspace to see what's available.
- It's usually a good idea to first check that the function arguments (here `parameters`) were called in correctly:





```
parameters
```

- That seems ok. If we hit `n` a few times we can check whether each line is coming up ok:

```
n  
beta0
```

- Note that the line `browser` displays is the line that is about to be run.



```
Console ~/    
Next   Continue Stop  
> neglikfun(c(1,-0.1))  
Called from: neglikfun(c(1, -0.1))  
Browse[1]> parameters  
[1] 1.0 -0.1  
Browse[1]> n  
debug at #3: beta0 <- parameters[1]  
Browse[2]> n  
debug at #4: beta1 <- parameters[2]  
Browse[2]> n  
debug at #5: daty0 <- y == 0  
Browse[2]> beta0  
[1] 1  
Browse[2]> |
```

Debugging the likelihood

- We should see that all the lines are working as expected, until we get to this one:

```
loglikelihood
```

- Somewhere in this line there is a mistake in. Let's check each bit:

```
sum(log(- (1/(1 + exp(-beta0 - beta1 * x[daty0])))))
```

- This sum is wrong, I have log of negative values - I've missed out a 1!

```
sum(log(1 - (1/(1 + exp(-beta0 - beta1 * x[daty0])))))
```

```

Browse[2]> beta0
[1] 1
Browse[2]> n
debug at #6: daty1 <- y == 1
Browse[2]> n
debug at #7: loglikelihood <- sum(log(1/(1 + exp(-beta0 - beta1 * x[daty1])))) +
sum(log(-(1/(1 + exp(-beta0 - beta1 * x[daty0])))))
Browse[2]> n
debug at #9: return(-loglikelihood)
Browse[2]> loglikelihood
[1] NaN
Warning message:
In log(-(1/(1 + exp(-beta0 - beta1 * x[daty0])))) : NaNs produced
Browse[2]>

```

- You can type Q to exit browser.

Other debugging tools

- Beyond browser
- trace and untrace
- traceback
- Setting breakpoints
- More advanced tools

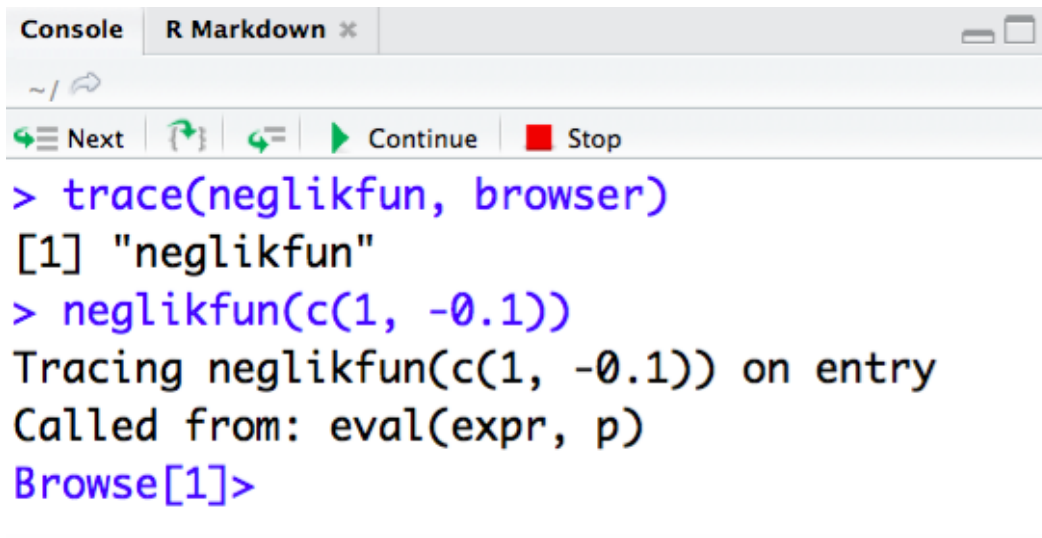
Beyond browser

- **browser** is more helpful than putting **print** commands everywhere, but it still involves fiddling around inside your functions.
- A slightly more elegant way of debugging involves calling **debug(neglikfun)** which will call **browser** every time you run the function. You can stop it by calling **undebbug(neglikfun)**.
- If you think the bug is only small and you'll fix it immediately (optimistic!) you can run **debugonce(neglikfun)** which will only call **browser** the first time you run the function.
- If you're in a loop and only want to call **browser** once, say, at $i = 20$, you can use **if(i == 20) browser()**
- Remember to hit Q when you're finished with **browser**.

trace and untrace

- **trace** gives you the opportunity to call a function when you enter another function.
- The general syntax is **trace(f, t)** where **t** is the function you want to call (e.g. **browser**) and **f** is the function you want it to be called from.

```
trace(neglikfun, browser)
neglikfun(c(1, -0.1))
```



- Of course this is essentially what `debug` is doing, but with far more flexibility.
- We can remove the trace with `untrace(f)`.

traceback and debugger

- If your code has already crashed and you want to see where, try `traceback()`.
- This will tell you which function crashed and which functions were called before it.
- You can even more information if you have set `options(error = dump.frames)`.
- Once you've done this you can type `debugger()` after the crash and will be able to explore the environments of the different previous functions
- Another is `options(error = recover)` which automatically enters `debugger` when an error is encountered.
- Switch all this off with `options(error = NULL)`.

Setting breakpoints

- Usually, you would put `browser` at the beginning of your function, but often this is unhelpful, as you know the bug occurs later in the code.
- In such cases you might set a breakpoint at certain locations where you think things are about to go wrong. One way to do this is simply to put `browser` at these points.
- Another way is use the `setBreakpoint` function. This works as `setBreakpoint(filename, linenumber)` where `filename` is the name of the .R file which contains your function and `linenumber` is the line at which you want to call `browser`
- An example. Suppose you had a call to `browser` and were currently at line 12 of your file `work.R` and you wanted the next `browser` call at line 28. Instead of exiting the function and inserting `browser` there you can just type `setBreakpoint(work.R, 28)` and hit `c`.

Final notes about debugging tools

- Take a look at the Debug menu in RStudio which will allow you to step through code and add breakpoints easily. More information can be found at: <https://support.rstudio.com/hc/en-us/articles/>

205612627-Debugging-with-RStudio

- There is a package **debug** that includes many more advanced tools for debugging R code.
- If you're dealing with random numbers, use **set.seed(x)** where **x** is some fixed number (e.g. 123) at the start of your code/function so that the random simulations repeat every time you run the code.
- Don't forget the obvious - brackets! More often than not the reason it's not working is because brackets aren't matching.
- If you see any **warnings** don't ignore them - it usually means something went wrong.

Extended example



Debugging findruns

Debugging findruns.

- Recall our function for finding runs of ones (lecture 2). This is a slightly broken version:

```
findruns <- function(x, k){  
  n <- length(x)  
  runs <- NULL  
  for(i in 1:(n - k)) {  
    if(all(x[i:i+k-1] == 1)) runs <- c(runs, i)  
  }  
  return(runs)  
}
```

- Can you spot where the bug(s) are?
- Suppose I had it stored in the file **findruns.R**. I can run:

```
source('findruns.R')  
y <- c(1, 0, 0, 1, 1, 1, 0, 1, 1)  
findruns(y, 2)
```

```
## [1] 3 4 5 7
```

- This is wrong!

Debugging findruns.

Let's try debug:

```
debug(findruns)  
findruns(y, 2)  
y
```

- That seems ok - it's read in y
- Step through a few more times:

```
n  
n  
print(n)
```

- Note **print(n)** rather than **n**. It seems to have got this correct...

```

Console R Markdown x
~/
Next
Continue Stop

Browse[2]> y
[1] 1 0 0 1 1 1 0 1 1
Browse[2]> n
debug at findruns.R#2: n <- length(x)
Browse[2]> n
debug at findruns.R#3: runs <- NULL
Browse[2]> print(n)
[1] 9
Browse[2]> |

```

```
n
n
i
```

- Check here that it's selecting the right parts - on the first loop it should be `x[1:2]`

```
x[i:i + k - 1]
```

- No - wrong!

```
i:i + k - 1
```

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

- We've missed a bracket - should be `i:(i+k-1)`!

```
i:(i + k - 1)
```

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

```

Console R Markdown x
~/
Next
Continue Stop

Browse[2]> n
debug at findruns.R#4: for (i in 1:(n
- k)) {
  if (all(x[i:i + k - 1] == 1))
    runs = c(runs, i)
}
Browse[2]> n
debug at findruns.R#5: if (all(x[i:i +
k - 1] == 1)) runs = c(runs, i)
Browse[2]> i
[1] 1
Browse[2]> x[i:i + k - 1]
[1] 0
Browse[2]> i:i + k - 1
[1] 2
Browse[2]> i:(i + k - 1)
[1] 1 2
Browse[2]> |

```

Did that fix it?

```
findruns <- function(x, k){  
  n <- length(x)  
  runs <- NULL  
  for(i in 1:(n - k)) {  
    if(all(x[i:(i+k-1)] == 1)) runs <- c(runs, i)  
  }  
  return(runs)  
}
```

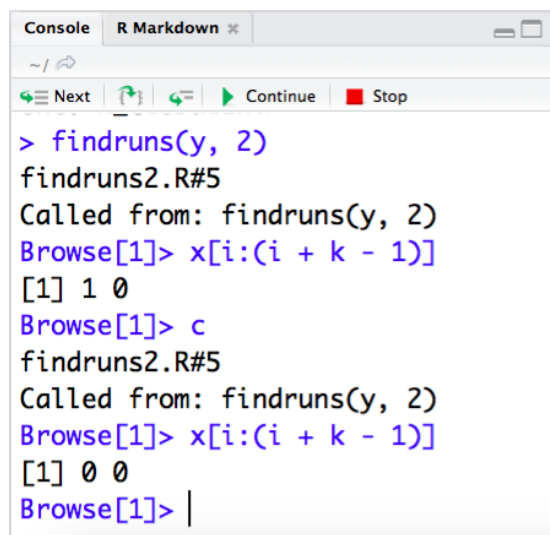
```
source('findruns2.R')  
findruns(y, 2)
```

```
## [1] 4 5
```

- No! It should be 4, 5 and 8
- Let's set a breakpoint inside the loop (line 5):

```
setBreakpoint('findruns2.R', 5)  
findruns(y, 2)  
x[i:(i + k - 1)]  
c  
x[i:(i + k - 1)]
```

- It seems to be reading in the right pairs of data - let's set a browser at $i = 8$.



```
Console R Markdown x  
~ / ↩  
⏮ Next ⏪ ⏩ Continue ⏹ Stop  
> findruns(y, 2)  
findruns2.R#5  
Called from: findruns(y, 2)  
Browse[1]> x[i:(i + k - 1)]  
[1] 1 0  
Browse[1]> c  
findruns2.R#5  
Called from: findruns(y, 2)  
Browse[1]> x[i:(i + k - 1)]  
[1] 0 0  
Browse[1]> |
```

Final steps

```
findruns <- function(x, k){  
  n <- length(x)  
  runs <- NULL  
  for(i in 1:(n - k)) {  
    if(i == 8) browser()  
    if(all(x[i:(i+k-1)] == 1)) runs <- c(runs, i)  
  }  
}
```



```
    return(runs)
}
```

```
source('findruns3.R')
findruns(y, 2)
```

```
## [1] 4 5
```

- Why isn't it entering **browser**?
- The upper limit of the loop is $n - k = 9 - 2 = 7$. It should be 8.
- We need to change the loop from `i in 1:(n-k)` to `i in 1:(n-k+1)`

Final steps

```
findruns <- function(x, k){
  n <- length(x)
  runs <- NULL
  for(i in 1:(n - k + 1)) {
    if(all(x[i:(i+k-1)] == 1)) runs <- c(runs, i)
  }
  return(runs)
}
```

```
source('findruns4.R')
findruns(y, 2)
```

```
## [1] 4 5 8
```

- It works!

Lessons from this week

- You can solve many problems by structuring your code well in the first place - write small functions and lots of comments.
- Learn how to use **browser** - it makes debugging far easier.