Assignment 1 Feedback

STA 242

April 24, 2015

Formatting Requirements

Please make sure you do the following in your report:

- Include the URL of your Bitbucket repository.
- Put margins on every page, including source code.[1]
- Use a white or light-colored background for source code.
- Format tables neatly (no raw output).

[1] Otherwise there's nowhere to write feedback.

Statistical Reporting

Below are some suggestions for writing statistical reports.

- When considering a statistical model or test, ask yourself:
  - What is the population?
  - What are the assumptions?
  - Are the assumptions reasonable for the data? Why?
  - Does it reveal anything a well-designed plot wouldn't reveal?
  - Does it have practical value for the data (e.g., prediction)?

  You should answer these questions in your report for any statistical procedure you decide to use.
- Generally, there should be *at least* 3 sentences discussing each figure. If you have a hard time writing this much, you should reconsider whether the figure is really important to your analysis.
- At the end of your report, briefly discuss the limitations of your analysis and how it could be improved or extended.
- Use colors and styles that are visually distinct even when printed in black and white. The package `RColorBrewer` is helpful for choosing colors.
- Put labels *with units* on all plot axes. The labels should be in plain English, not variable names from your code.
- Use a spell-checker. You won't lose points for spelling errors, but they definitely make you look lazy and unprofessional.
- Use external sources to enhance your analysis. Make sure to cite them!

Code Design

Make sure you follow all of the advice from Duncan's code review. Below are a few more suggestions, based on things I saw while grading Assignment 1.

Fusing Loops

Suppose we want to apply two functions to every column of a data frame. We could try something like this:

```
data = lapply(data, clean)
data = lapply(data, set_types)
```

However, this code is inefficient. If there are 10 columns, it will loop 20 times. The problem gets worse when there are more columns or more functions to be applied. Instead of using a separate loop for each function, *fuse* the loops:

```
data = lapply(data, function(col) {
    col = clean(col)
    set_types(col)
})
```

You should always be on the lookout for loops that can be fused in your code.

Avoiding Redundant Computations

Consider the function

```
function(text = "Everything is awesome!") {
    if(strsplit(text, " ")[[1]][1] == "Nothing"){
        return(":(")
    }

    text = strsplit(text, " ")[[1]]
    text[3] = "cool when you're part of a team!"
    return(paste(text))
}
```

The call to `strsplit()` appears twice with the exact same arguments. Each time, R has to recompute the split string, but the result is the same. This is not efficient. A better strategy is to store the split string in a variable:

```
function(text = "Everything is awesome!") {
    text = strsplit(text, " ")[[1]]
```

```
    if(text[1] == "Nothing"){
        return(":(")
    }

    text[3] = "cool when you're part of a team!"
    return(paste(text))
}
```

Avoiding redundant computations is especially important when using time-consuming functions such as `gsub()`.

Combining Functions

Many of you put a lot of effort into writing short functions, which is great. However, I occasionally saw something like this:

```
clean1 = function(text) {
    # Strip leading and trailing whitespace.
    text = gsub('^\\s*|\\s*^', '', text)

    # Convert to title case.
    text = strsplit(text, ' ')[[1]]
    first_letters = toupper(substring(text, 1))
    other_letters = substring(text, 2)

    paste(first_letters, other_letters, sep = '', collapse = ' ')
}

clean2 = function(text) {
    # Replace non-breaking spaces.
    text = gsub('\u00A0', ' ', text)

    # Strip leading and trailing whitespace.
    text = gsub('^\\s*|\\s*^', '', text)

    # Convert to title case.
    text = strsplit(text, ' ')[[1]]
    first_letters = toupper(substring(text, 1))
    other_letters = substring(text, 2)

    paste(first_letters, other_letters, sep = '', collapse = ' ')
}
```

These two functions are nearly identical, and could be combined very easily by adding a new parameter:

```
clean = function(text, replace_nbsp = FALSE) {
    # Replace non-breaking spaces.
    if (replace_nbsp)
        text = gsub('\u00A0', ' ', text)

    # Strip leading and trailing whitespace.
    text = gsub('^\\s*|\\s*^', '', text)

    # Convert to title case.
    text = strsplit(text, ' ')[[1]]
    first_letters = toupper(substring(text, 1))
    other_letters = substring(text, 2)

    paste(first_letters, other_letters, sep = '', collapse = ' ')
}
```

However, this new function isn't quite perfect—it's trying to do several unrelated things (its vague name "clean" is a hint there's a problem). We can improve the code by splitting it into two functions again, dividing the code according to its purpose:

```
strip = function(text, replace_nbsp = FALSE) {
    # Replace non-breaking spaces.
    if (replace_nbsp)
        text = gsub('\u00A0', ' ', text)

    # Strip leading and trailing whitespace.
    gsub('^\\s*|\\s*^', '', text)
}

to_title = function(text) {
    # Convert to title case.
    text = strsplit(text, ' ')[[1]]
    first_letters = toupper(substring(text, 1))
    other_letters = substring(text, 2)

    paste(first_letters, other_letters, sep = '', collapse = ' ')
}
```

Simplifying `if` Statements

When using an `if` statement, try to minimize the code inside. This makes it easier to see what the `if` statement is really doing. For example, compare

```
number = if (text == 'E.T. phone home.')
    '555-4242'
else
    '555-2287'

cat(paste0('Dialing ', number))
dial(number)
```

where it's immediately clear that the `if` statement sets the phone number, to

```
if (text == 'E.T. phone home.') {
    cat('Dialing 555-4242.\n')
    dial('555-4242')
} else
    cat('Dialing 555-2287.\n')
    dial('555-2287')
}
```

where it's hard to see what the difference is. Clarifying your code this way can also make it easier to reason about and debug. When an `if` statement handles a special case, you should also include a comment explaining why it's necessary.

Using `any()` and `all()`

A few of you did something like this:

```
if (sum(x == 5) > 0) {
    # do something...
}
```

This code works, but it's a little inelegant, and it probably took you a second to understand what it does. We can make it clearer with `any()`:

```
if (any(x == 5)) {
    # do something...
}
```

Similarly, you should also use the `all()` function where it makes sense.

Writing Readable Code

Consider the 3 equivalent `if` statements

```
if (is.na(x) == TRUE) x = 1

if (is.na(x) == 'TRUE') x = 1

if (is.na(x)) x = 1
```

The first isn't wrong, but checking `== TRUE` is redundant and doesn't make the expression any clearer. The second *is* wrong—but works anyways (why?)—and you should **never** use it. The third is the most parsimonious, and is the expression you should prefer. The idea here is to try to write concise code the reads nicely. For instance, if you wanted to negate the condition, you should use

```
if (!is.na(x)) x = 1
```

which reads as "if `x` is not `NA`" rather than

```
if (is.na(x) == FALSE) x = 1
```

which reads as the less intuitive "if `x` is `NA` is false."