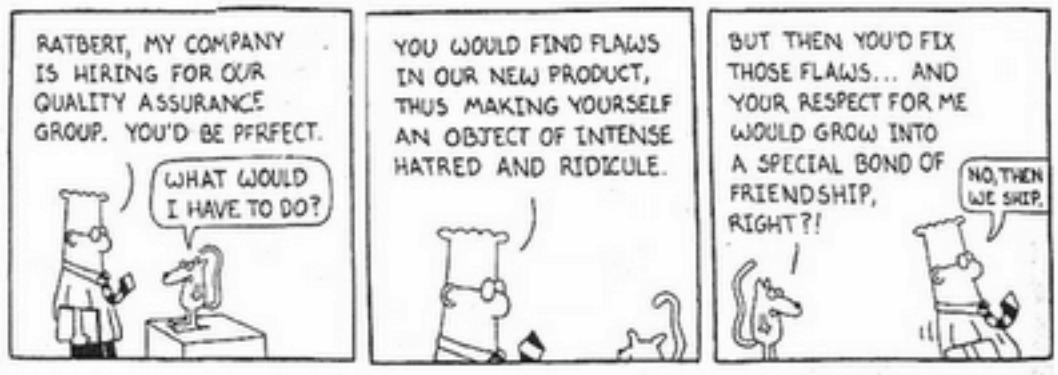
Unit Testing in R: A Quick Tutorial to Improve Efficency in Debugging

Victor Xu

October 31, 2017

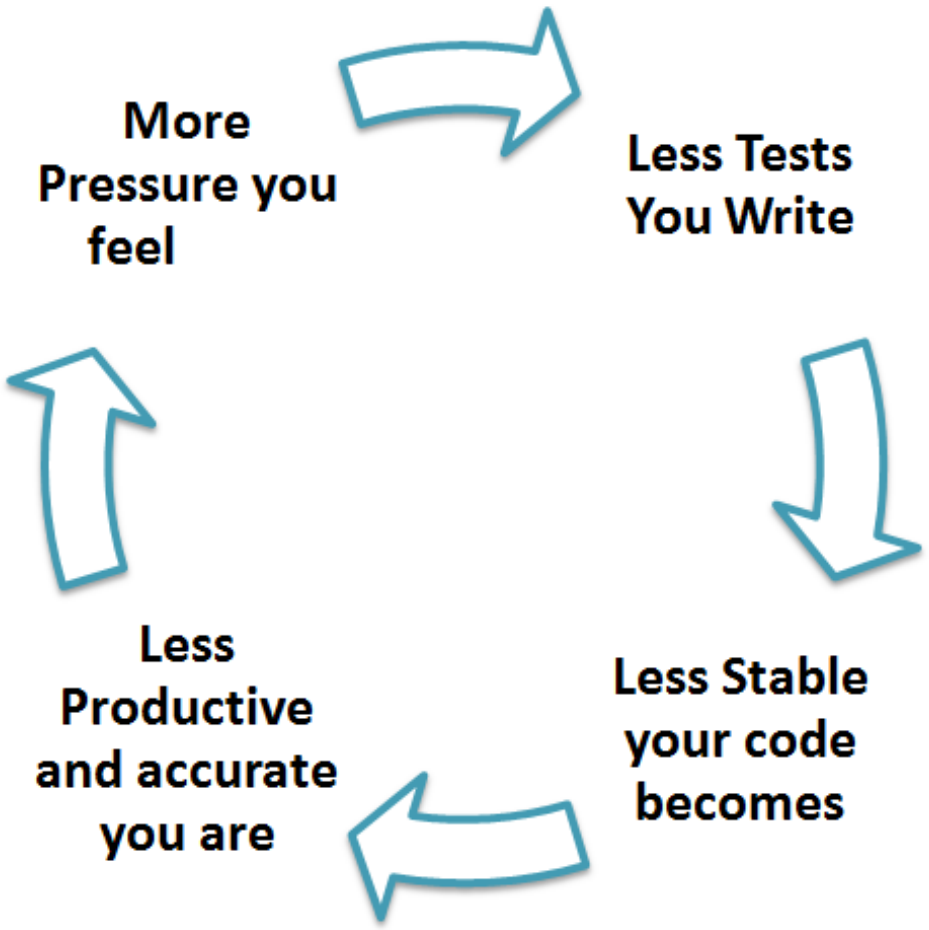
Introduction

In the software development world, the concept of testing is well developed and crucial to the success of larger projects. As an analogy, let's observe elementary school students learning addition and subtraction before tackling multiplication and division. In the end, the student is expected to combine these arithmetic skills to solve complicated equations and functions. But what happens if this student doesn't learn addition, and skips ahead? We can assume that all questions involving addition as a procedure would be answered incorrectly. This would result in great anxiety for the teacher and student, perhaps unaware of where the source of the problem is stemming from. Furthermore, going back to address this miscommunicated concept ('addition') and re-checking everything would be embarrassing and hugely inefficient. This story gets at the heart of unit testing: programmatic tests (usually simple, low level, and sometimes automated) which we can procedurally write to evaluate bits ("units") of our code for functional correctness. When applied correctly, it can save us from the painful process of going through mountains of code, wondering why cryptic error messages are popping up at the end.



Comic Strip on Debugging Fustration

This is essentially my motivation for writing this post - to explore how unit testing can be applied to the R programming language using three practical examples (one of which is visual and relates to plots/outputs). Because unit testing is a critical concept introduced in most introductory computer science courses (i.e. the infamous 'CS61A' taught at the University of California, Berkeley), I thought it would be a practical topic to approach in this post; especially as I become more proficient in the R language and attempt longer/more difficult projects in the near future.



The Importance of Unit Testing

More importantly, this is a topic which has yet to be addressed in Statistics 133 - the course I am submitting this post for. This presents an original opportunity to go beyond what I've so far learned from lectures, labs, and tutorials. So read on! Whether you're a fellow student in Stat 133, aspiring data scientist, or just curious about expanding your knowledge - this post will hopefully be useful in providing some concrete examples about unit testing, in a world where programming is increasingly interwoven with all types careers and subjects. Let our exploration begin!

Background

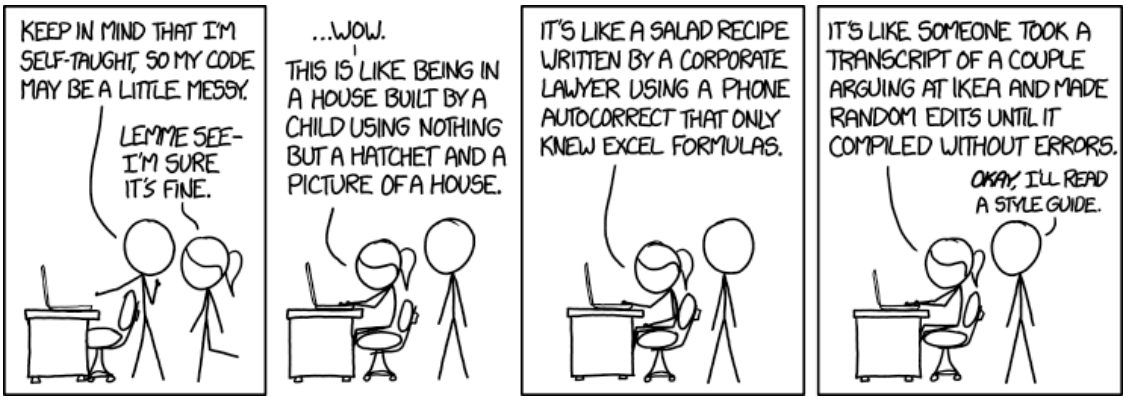
In our post, let's start by reviewing the goals and methodology for constructing a unit test. For our purposes, let's assume that we want unit tests to also be functional tests. That is, the tests should verify if our code actually does what we need it to do. We can construct unit tests for many parts of the code: from inputs (Are we importing files correctly? Are the proper functions or packages in R being called? Is the format of the data being imported correct?), output (Is the result similar to what we expect?), calculations (Are we using the right algorithms or equations?), results, and so forth.

Of course, knowing where to place unit tests requires some specificity. Depending on what the code is being used for, you should consider framing questions ahead of time. For instance:

- Input stage: What happens if a dataset contains 'NULL' values or missing values?
- Calculation stage: Will all of the values need to be a certain unit? Are conversions necessary?
- Output stage: How should outliers be handled? Will it be included in the final output - whether numerically or visually (i.e. plots)?

Your responses to these questions will also prioritize different factors, again dependent on the project's needs. Common factors may include:

- Correctness (Does the result match your expectations?)
- Efficiency (How is the performance of the code? Does it run in acceptable time?)
- Style (Is the code overly long? Does this impact the efficiency or ability to make quick modifications in the future?)



Factors to Consider: Style & Efficiency

Based on how you respond to these questions, you can now construct the unit tests to:

1. Formulate tests to catch these obvious errors from the various stages of the code (starting from the beginning and progressing chronologically)
2. Write these unit tests and run w. data
3. If bugs appear in a 'unit', now fix it
4. Re-run the code and check that everything works

To construct the unit test, we also need to know where to put it! Generally, the tests we write will live in two possible places:

1. If the test correlates with a function and needs to be run each time the function gets called, then we can place the test directly with the function. These are defined as "run time" tests. Run-time tests produce small assessments and can throw a warning or error message.
2. Alternatively, tests can be in their own file if we are checking at-large if everything works after modifications to the code has been made. These are defined as "test time"

tests. Test-time tests may use assert statements (functions) to check the validity of a condition we want.

Now let's proceed with some examples covering these concepts.

Example 1:

Following example of a simple, "run time" unit test is below. Note that we are using the assert statement stopifnot() in this example, to stop the program if the values don't equal what we want from the test.

```
This would be in a script in an actual project

#First: defining the variables x and y
x <- 10
y <- x * 2

#Now we are checking if y is equal or greater to a value, say 5.
#Because this statement holds, we do not stop the code - continues to run.
stopifnot(y >= 5)

#Let's say more changes were made to x and y
x <- x * 0
y <- x - 1

#Now that y is no longer greater than 5, an error is thrown (Error: y >= 5 is
not TRUE)
#The condition is no longer met and the code haults.
stopifnot(y >= 5)
```

Discussion/Reflection:

This simple example introduces us to the built-in stopifnot() assert function, which is quite useful when testing whether conditions will continuously be met as the code progresses. We see that when the assert function fails, a clear error message is thrown, and we are able to quickly fix whatever problem has gone wrong.

Of course, when the statement is no longer needed, we can also comment it out from the code. We can do this by adding '#' to the front of the line, such as: # stopifnot (), if we are no longer interested in testing this particular condition; giving us some flexibility. Overall, I can see this example being useful in the input and calculation stages of a project. For instance, stopifnot certain values are still 0 (or empty), could be a useful test to impose during the data processing stage of importing datasets. In the calculation stage, this function could also gauge if something serious has gone awry (i.e. not in the same order of magnitude as expected output, etc.).

Other helpful functions include, but are certainly not limited to: - checkequals() #For instance, whether conversions are equal between absolute zero and degrees C/F - checkequalsnumeric() #Same as above, but ensures the values are also of 'numeric' type

Example 2:

The following example covers a "test time" test, which is more conceptual. Ideally, we would write these tests before we write our actual function and code. We use the package 'testthat' and 'devtools'to create the directory structure needed for the testing. This example will summarize the process and include snipits of code for how this would look like in an actual project:

Firstly, the code chunk below assumes that there are multiple functions in file "sample.R" that you want to test. We would want to create a directory called tests to store all our test cases. In tests, we would create files like 1.R to contain the first set of tests, and so forth. The function expect_that() is used in our example below:

```
install.packages('testthat')

expect_that(1 ^ 1, equals(1))
expect_that(3 ^ 2, equals(9))

expect_that(1 + 1 == 2, is_true())
expect_that(3 == 1, is_false())

expect_that(2, is_a('numeric'))

expect_that(print('Hello World!'), prints_text('Hello World!'))

expect_that(log('Hello World!'), throws_error())
```

To run these tests, we would need to create a file called run_tests.R to act as a test suite and invoke all of the tests we wrote. This would automate the testing process. Here's an example of what this might look like:

```
library('testthat')

source('sample.R')

test_dir('tests', reporter = 'Summary')
```

Discussion/Reflection:

We see that with "test time" tests, we think of possible holes (corner cases) in our own code, which might translate into bugs, and attempt to catch them. After an error appears, we go back, address the problem, and re-run to see if everything now passes. This process does require some more thinking than say, the previous example with our "run time" test. We've also introduced the testthat() package, with functions it should include, such as expect_that(), which can be very helpful when testing the validity of our calculations.

To opt out of a particular test, note that we can use the function skip() to move onto the next test we've written.

Example 3:

This final example borrows concepts from the first two cases, and applies it to unit testing on images. Although a little more complicated, we can use unit tests to check if the parameters of our graphics match our expectations.

In this example, we are calling on an open source package which contains images we can use to compare to one another.

The code can be explained as follows: 1. We create some plots, then we use the function getFingerprint() to extract fingerprints for each plot (summary/essential parts) and compare them.

```
install.packages(devtools)

devtools::install_github("MangoTheCat/visualTest")
library(visualTest)

png(filename = "test1.png")
img()
dev.off()

png(filename = "test2.png")
plot(1:11, col="red")
dev.off()

getFingerprint(file = "test1.png")
getFingerprint(file = "test2.png")

isSimilar(file = "test2.png",
          fingerprint = getFingerprint(file = "test1.png"),
          threshold = 0.1
```

Discussion/Reflection:

Because the two images are not the same, the isSimilar() function test will fail. We will evaluate 'FALSE' as the expected output. More about this open source package can be found in the link below: http://www.mango-solutions.com/wp/products-services/r-services/r-packages/visualtest/

Conclusion & Take-Away:

Congratulations!

By making it to the end of this post, you've obtained a deeper understanding of how unit tests can be constructed in R. Implicitly, by going through several practical examples of unit tests integrated into various stages of code, I hope I've reduced the anxiety and enigma surrounding how to set-up a unit test in the first place. With these differing examples in mind, you should feel confident to include such tests at various segments of longer projects to ensure correctness, efficiency, and style.

Explicitly, we've iterated on the reasons why unit tests are important - and my aim was to let you see how this isn't a process which requires extended effort. Practicing to include unit tests habitually is the sign of a wise programmer, and this something which can easily picked up. Again, the point of unit testing to make the process of debugging (if currently - gasp - left until the very end) less painful and ultimately time-consuming. By organizing our tests and logically thinking through our code, we are dramatically reducing the possibility of sloppiness and larger mistakes from ruining our code - like the analogical child from the beginning who skipped over addition. We can only sympathize with the poor soul, who, ignores all the unit tests raised in our examples, only to find catastrophic errors at the end. Even if unit testing is something which requires some practice to master, it can only help, never hinder.

Like medicine for the mind, a Chinese proverb rings true. "Xian Ku Hou Tian" . "First Bitter, But Then Sweet"

References:

1. https://www.r-bloggers.com/unit-testing-with-r/
2. http://r-pkgs.had.co.nz/tests.html
3. http://www.johnmyleswhite.com/notebook/2010/08/17/unit-testing-in-r-the-bare-minimum/
4. https://www.garysieling.com/blog/unit-testing-with-r
5. https://www.johndcook.com/blog/2013/06/12/example-of-unit-testing-r-code-with-testthat/
6. https://stackoverflow.com/questions/31038709/how-to-write-a-test-for-a-ggplot-plot
7. https://stackoverflow.com/questions/30246789/how-to-test-graphical-output-of-functions
8. https://www.youtube.com/watch?v=8G6bBum3O9A
9. https://www.youtube.com/watch?v=CAy0udiWwmg
10. https://cran.r-project.org/web/packages/RUnit/vignettes/RUnit.pdf