mutant: mutation testing for R

Scott Chamberlain

2021-01-29

Signatories

Project team

Scott Chamberlain, rOpenSci Co-founder and Technical Lead, University of California, Berkeley

Contributors

Consulted

- Gábor Csárdi, RStudio
- Maëlle Salmon, rOpenSci
- Karthik Ram, rOpenSci
- . . .

The Problem

Unit tests are used to ensure that code works as intended. Code coverage is usually used as a tool to indicate how “good” your tests are. However, even 100% code unit test coverage does not sufficiently ensure the code is robust to current errors due to user inputs, or future errors due to changes in the source code. It’s easy to see how code coverage falls short:

A function that gives a boolean if a number is greater than 10:

```r
greater_than_ten <- function(x) {
  x > 10
}
```

A unit test for the `greater_than_ten()` function:

```r
testthat::test_that("greater_than_10 works as expected", {
  expect_true(greater_than_ten(11))
  expect_false(greater_than_ten(9))
})
```

Code coverage would be 100%, but by looking at the code there’s clear cases where the above unit tests would not be sufficient. For example, what if a user passes a type that is a not a number? Booleans can be passed in and will be treated as 0 (`FALSE`) or 1 (`TRUE`), but it would make more sense to throw an error instead most likely.

Whereas unit testing checks that a module's code works as expected, mutation testing does the same for those unit tests. Mutation testing was first proposed by Richard Lipton in 1971, and first implemented in 1980. However, due to the computationally intensive nature of mutation testing, the technique wasn't used more widely until computer power improved. In mutation testing, the code is broken down to an abstract syntax tree (AST), then a "mutant" is created by changing a single operator (e.g., swap a boolean). The idea is that the unit tests should now fail against the mutant; if a test fails then the mutant is said to be "killed". When unit tests do not fail against a mutant the mutant has "survived" and indicates a problem.

The main goal of mutation testing is to improve unit tests. It does this by exposing flaws in current tests, as well as suggesting new tests through surviving mutants (source code changes that unit tests do not catch).

Mutation testing entails a fair amount of challenges and complexity. First, there's the limitations of R. In the R programming language we do not have an easy way to create and work with an AST of an R script or package. There are some R tools (e.g., some functions in rlang) but nothing that allows easy modification and re-creation of code from an R file. In Python or Ruby, for two examples, there's easy to use tools to break down a script into an AST, modify that AST, and rewrite to disk (Python docs, see refs).

Second, mutation testing can be time consuming. For example, if one run of unit tests for a package takes one minute, then for a mutation testing run we'd multiply one minute by dozens or hundreds of runs, one for each mutant. Of course this can be sped up by parallelizing, among other tricks. For large packages, running mutation testing on a cloud hosted CI will be the easiest solution.

Implementing queues for running all mutants that work across operating systems, and with good failure behavior, will potentially be challenging, but very much solvable. MORE EXPLANATION

The proposal

We propose to finish the mutant R package. mutant will learn from the success of the usethis package, with intuitive functions with clear use cases, as well as helper functions to setup required infrastructure mutant requires. There are no other mutation testing tools in R that I know of.

The mutant package will only work with the testthat package at first because it is the most popular testing package: ~5540 reverse dependencies for testthat compared to 129 for tinytest, and 33 for testit. Mutant should be easily extended to the other two testing packages.

To make manipulating R ASTs easier for the mutation testing, a companion project astr (https://github.com/sckott/astr) will help us manage and modify R AST's.

We'll address the time consuming nature of mutation testing with various approaches. I'll leverage the callr package to run multiple and parallel R processes in the background at the same time. In addition, I'll make clear documentation on when to use mutation testing; for example, you don't need to run mutation tests after every code change as you might for unit tests.

A well done queueing system will be important. The liteq package will be used to implement the queueing system - which uses a portable SQLite based system that will work on all operating systems. The R6 package will be used to implement the queueing logic.

Mutation testing in the R community will not be used as widely as unit testing - not every one that uses unit testing will use mutation testing. Those that will find mutation testing most useful will likely be: those in enterprise that place a higher value on code doing what it says it will do; and open source package maintainers with heavily used/depended upon packages. With time mutation testing will likely become more widely adopted in the R community as the benefits become clear, and the cost of doing mutation testing (time) decreases.

The following is an example using pseudo-code. Although the mutant package has many of the pieces, it does not yet have working code to show an example.

Code in a file `R/foo.R`:

```r
less_than_ten <- function(x) {
  z <- x + 5
  z < 10
}
```

Create two mutants:

```r
less_than_ten <- function(x) {
  z <- x - 5 # changed + to -
  z < 7 # 10 changed to 7
}
```

Run mutant:

```r
library(mutant)
mutant::run()
```

Output:

```
Mutant killed: R/foo.R: line 2:5
  Mutator: ReplaceOperator
  -  z <- x + 5
  +  z <- x - 5

  Mutant survived: R/foo.R: line 10:27
  Mutator: SwapNumber
  -  z < 10
  +  z < 7
```

In the above example, one mutant was killed (good) and one survived (bad). That is, the unit tests for this function are adequate for the first mutation (no changes needed), but are not adequate for the second mutation.

Major pieces to complete

- Creating mutations: First, finish implementing abstract syntax tree (AST) tooling (astr package). Second, finish implementing the use of astr in a comprehensive set of tools in mutant for creating mutations.
- Implement a queueing system to be able to spin up R sessions, and then collect results from each session.
- Functions for reporting to the user results of test runs. Leveraging cli/symbols packages to create easy to interpret results
- Documentation
  - (edit this) From Maelle: what would it take for a package maintainer to get started with mutation testing their package? running a setup function only? more?

(Use a schematic of the workflow within the package?)

Dissemination

- License: is currently MIT, and there's no plans to change the license
- Code is on GitHub https://github.com/sckott/mutant, https://github.com/sckott/astr
- Blog posts, rOpenSci community call, conference talk, R Consortium blog post
- Submission to CRAN

Project plan

Start-up phase

Technical delivery

Eight weeks in total. 4 hours per day, 4 days per week (16 hrs/wk * 8 weeks = 128 hrs)

- Phase 1: Creating mutations: AST tooling and mutant package functions for creating mutants - package astr
  - Weeks 1-2 (32 hrs)
- Phase 2: Implement queueing tooling to be able to run many R sessions in parallel
  - Weeks 3-6 (64 hrs)
- Phase 3: Implement reporting tooling for showing results of mutation testing and pointing out where mutations occurred in the code and what the mutation may indicate
  - Week 7 (16 hrs)
- Phase 4: Write detailed package level documentation as well as detailed vignettes, including a tutorial for a very small hello world example to get users oriented, as well as a tutorial for a larger package with a lot of code and tests to demonstrate solutions for saving time, running on CI, and more
  - Week 8 (16 hrs)

Other aspects

Requirements

Programming time is the only requirement. All of the requested funds are for paying salaray of the one programmer.

People

Scott Chamberlain - rOpenSci Scott has implemented the beginnings of the mutant package, and is very familiar with the package. Scott has extensive experience in developing R packages, including very complex packages (for example, vcr https://github.com/ropensci/vcr).

Processes

The mutant project will use a Code of Conduct following the rOpenSci Code of Conduct https://ropensci.org/code-of-conduct/

Tools & Tech

Funding

I request a total of $16,000 to support 2 months work for myself, all of which will be salary.

- Salary: $16,000
- Total award: $16,000

Summary

Success

Definition of done

This project will be complete when a stable version of mutant is on CRAN and users are starting to give feedback - at that point it will be on its way to wider adoption.

Measuring success

Success will be measured by interest gauged via GitHub (stars, forks, opened issues), downloads from CRAN, as well as searching usage in GitHub repositories. Most importantly, I'd like to attract one or more additional maintainers to make sure the project is more sustainable moving forward - and because complex projects are easier with more maintainers.

Future work

Additional work beyond the scope of this proposal depends on user input of course. Some ideas in mind that may be worked on:

- Map file names to lines of code in those files and what unit tests test each of those lines of code. With this we could pinpoint and leverage the line of code in each mutant. Some of the prerequisites for this may exist already in the `covrpage` package (https://github.com/yonicd/covrpage).
- Make mutant runs faster by only running unit tests that are impacted by the mutant itself. This could dramatically speed up runs. We'd need the prerequisite discussed in the above bullet first to make this possible.
- The way R parses functions is not ideal. It discards comments, blank lines, etc. Which means that it's difficult to retain the structure of the original file function(s) come from. Right now, we're reading files, doing mutations, then writing completely new files - this in some cases won't work because we lose meaningful pieces that R's parsing does not retain. We can still implement mutant regardless, but better parsing of the language would help.

Key risks

As mentioned above, this project is rather complex, which could lead to the project taking longer than expected. However, there are parts of the project that can be reduced in scope to meet the timeline proposed. For example, we can implement a very minimal reporting setup so that basic information is present with looking sleek. In another example, there may be some operating systems the package may not work on when running R sessions in parallel; we can still move forward and solve those problems later.

The smaller niche of mutant relative to other projects, testthat for example, may relegate it to low usage. I think we can drive usage up with thorough documentation, as well as many blog posts describing usage.

Last, to make sure the project lasts a long time, I'll recruit co-maintainers as soon as possible and give them write access to the two repositories so they will know they're part of the team. In fact, there is already one contributor to mutant (https://github.com/sckott/mutant/graphs/contributors) who I am confident will contribute if I am actively working on the project.

References

Abstract Syntax Trees in Python https://docs.python.org/3/library/ast.html