662 Objective Correctness Report

Dohyun Lee, Maxwell Lovig, Sahil Singh

November 2022

Introduction

Our goal for this package was to create a grading algorithm that achieved the two goals

- Allow a user to have rough grades for assignments in as few lines as possible
- Allow a user to have nearly infinite control over the algorithm if desired

In many respects these two goals are orthogonal, to have one means we must relinquish the other. For our project, we tried to find an even balance between both. In order to achieve this, however, we had to simplify the types of submissions we can grade. After discussion, we decided to only allow the case of functional submissions. We feel that most problems we would want automatically graded reside in this domain. Even if a script case is required, Jay has a script that can handle the conversion from script problems to function problems.

Moving on to our methods, we need the user to have the following objects in R to be able to use our methods;

1. A list F of functions, each of which is a submission from a student
2. A singular "Correct" function C
3. a list T of test cases which the functions will be compared against

Once we have this we can use our default grading scheme to get rough grades for the user. Below we describe a high-level overview of our algorithm and indicate what parts a user can control over with the symbol $\star$

Let us consider one single $f \in F$ and a single $t \in T$, and analyze the first step in our algorithm. We take test case t and plug it into submission f, this will give us output $y_{f,t}$. We then compare this output with test case t plugged into the correct function C, this will give our output $\bar{y}_t$. By default, we then compare the outputs $y_{f,t}$ and $\bar{y}_t$ using the *all.equal* function in base R. This function will compare the outputs in many ways and return a list of differences (we will call them error codes for the remainder of the report) between $y_{f,t}$ and $\bar{y}_t$. There are some caveats to this:

- If the function f cannot produce an output, we assign the error code of "DNR" (Did Not Run). $\star$ A user can also define a maximal **timeLimit**, where if f takes longer then **timeLimit** to run it is also assigned "DNR"
- If *all.equal* returns no errors then we assign an empty list (i.e. no error codes)
- $\star$ If more robust tests are required then the user can write a list of custom comparisons, **customs** and a list of custom error codes, **ErrCodes** which can also be added to those produced by all.equals

We then create a matrix of dimension $\#F \times \#T$ where the i-th row and j-th column is a list of error codes between $y_{F[i],T[j]}$ and $\bar{y}_T[j]$. We denote this matrix as **mat**.

We then take **mat** and manipulate it in a manner to assign a score of "badness" to each student's submission. Depending on how much control the user wants, they have two options:

First, they can decide that they want the default scoring method. This method assigns equal badness to each error that could have been made. Essentially their "badness" score is the length of their error code list. If they have "DNR" then we flag their score with a -1[1].

Secondly, if they want to provide custom weighting scheme for each of the errors then they must have a list of all the unique errors made. Such a list can be found with the following code:

U <- unique(unlist(mat))

⋆ One can also create a custom vector **WErr**, in which **WErr**[i] is badness score for the error code U[i]. We then take each unique error $u \in U$ and assign a badness of making error u.

We also have a weighting of test cases, by default we consider each test case as equally important. ⋆ One can also create a custom vector **WCase**, in which $WCase$[i], is the weight of how important it is to get test case $T[i]$ correct

Once we have these materials we can then take the badness score for each student which is combined across all test cases[2]. We then have a badness vector where $b[i]$ is the cumulative badness for the i-th submission. Once we have this we can process the grading in two ways: curved or non-curved. Before we dive into that we have two commonalities between then

1. If submission i has $b[i] = 0$, then we give them a grade of 100
2. If submission i has $b[i] < 0$, then this code did not run and we give a grade of 0

Now that we have covered the commonalities, lets explain what happens when $b[i] > 0$.

In the non-curved case we have the grade for submission i, $g[i]$, to be

$$g[i] = \frac{\max_j(b[j]) - b[i]}{\max_j(b[j])}$$

In the curved case we have the grade for submission i, to be

$$g[i] = \Phi^{-1}_{80,5}\left(\frac{\max_j(b[j]) - b[i]}{\max_j(b[j])} \cdot .999 + .5 \cdot .001\right)$$

Which will either grade proportionally to badness (in the non-curved case) or grade on a normal curve centered at 80 with std deviation 5.

A Use Case

A Complete review of the capabilities of our function can be seen in the documentation of our code. Instead of redescribing our documentation, we provide a small toy example that will give an overview of the methods at play. Let's assume that we have gotten our list of submission functions (for this example it is of length 4), correct functions, and tests(for this example it is of length 2). We then run these through *checkEquiv* to get the error matrix

$$\mathbf{A} = \begin{bmatrix} (A) & (A,\ B) \\ (DNR) & (DNR) \\ (C) & (\ ) \\ (\ ) & (\ ) \end{bmatrix}$$

Where $(\cdot)$ represents the list structure and A, B, C, DNR are all error codes.

[1] See the discussion at the end, this may be a point of contention

[2] We give a very simple example in the Use Case Section

```
L <- list(
     list("A"), list("A", "B"), list("DNR"), list("DNR"), list("C"), list(), list(), list()
     )

A <- matrix(L, nrow = 4, byrow = T)
A

##      [,1]   [,2]
## [1,] list,1 list,2
## [2,] list,1 list,1
## [3,] list,1 list,0
## [4,] list,0 list,0

B <- scoreFuncs(A)
```

Let's describe what is going on in *scoreFuncs*. First, as the user gave no defaults, we find all the unique errors made. Clearly, in this toy example, the vector is some permutation of:

$$[A \quad B \quad C \quad DNR]$$

Our code generates this permutation (it unlists column-wise first).

```
unique(unlist(A))

## [1] "A"   "DNR" "C"   "B"
```

Thus we have

$$\mathbf{U} = [A \quad DNR \quad C \quad B]$$

We now binarize each entry of $\mathbf{A}$ according to $\mathbf{U}$, this will produce the following matrix

$$\mathbf{B}_3 = \begin{bmatrix} [1,0,0,0] & [1,0,0,1] \\ [0,1,0,0] & [0,1,0,0] \\ [0,0,1,0] & [0,0,0,0] \\ [0,0,0,0] & [0,0,0,0] \end{bmatrix}$$

```
B[[3]]

##      [,1]      [,2]
## [1,] numeric,4 numeric,4
## [2,] numeric,4 numeric,4
## [3,] numeric,4 numeric,4
## [4,] numeric,4 numeric,4

B[[3]][1,2]

## [[1]]
## [1] 1 0 0 1

B[[3]][4,1]

## [[1]]
## [1] 0 0 0 0
```

We then take $\mathbf{B_3}$ and do a pseudo-tensor operation, defining our case-dependent badness as

$$\mathbf{B}_2[i,j] =< \mathbf{B}_3[i], \mathbf{WErr}[i] >$$

Where **WErr** is the error code weights (we use the default $\mathbf{WErr} = [1\ -1\ 1\ 1]$) and $< \cdot, \circ >$ is the euclidean inner product. We have our $\mathbf{B}_2$ as

$$\mathbf{B}_2 = \begin{bmatrix} 1 & 2 \\ -1 & -1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}$$

```
B[[2]]
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]   -1   -1
## [3,]    1    0
## [4,]    0    0
```

To get the cumulative badness $\mathbf{B}_1$, which we will use to grade the submissions, we have

$$\mathbf{B}_1 = (\mathbf{B}_2)(\mathbf{WCase})^T$$

Where **WCase** is the weighting of each test case (we use default $\mathbf{WCase} = [1, 1]$). Giving us:

$$\mathbf{B}_1 = \begin{bmatrix} 3 \\ -2 \\ 1 \\ 0 \end{bmatrix}$$

```
B[[1]]
```

```
##      [,1]
## [1,]    3
## [2,]   -2
## [3,]    1
## [4,]    0
```

We now pass these cumulative badness scores into our grading method. We have the grades if we used the non-curved or curved methods below.

```
grades <- getGrade(B[[1]], curved = F)

matrix(c("Sub 1", "Sub 2", "Sub 3", "Sub 4", round(grades)), ncol = 2)
```

```
##      [,1]    [,2]
## [1,] "Sub 1" "0"
## [2,] "Sub 2" "0"
## [3,] "Sub 3" "67"
## [4,] "Sub 4" "100"
```

```
grades <- getGrade(B[[1]], curved = T)

matrix(c("Sub 1", "Sub 2", "Sub 3", "Sub 4", round(grades)), ncol = 2)
```

```
##      [,1]    [,2]
## [1,] "Sub 1" "64"
## [2,] "Sub 2" "0"
## [3,] "Sub 3" "82"
## [4,] "Sub 4" "100"
```

For a more in-depth example see our code.

Commentary

Our code allows the user, the instructor, to automate grading student R script submissions so long as the answer is in the form of a function. In our function *checkEquiv* we made good use of the base R function all.equal() to get our n-by-m matrix that represents the errors by the i-th student on the j-th test case – the initial idea was to use the testthat or the gradeR package to evaluate the errors made by the student and somehow weight the errors into a grade, but we decided that our *checkEquiv* function is a more reproducible method.

One could argue that the use of nested for-loops and various if-statements is inefficient for this problem. However, the average size of a course is not that big – even if this code were used on a course with 500 students and the runtime of our algorithm took 10 minutes it would still be a lot more time and cost-efficient than hiring multiple graders and making them run through each script and assigning grades over the span of many days. On that note, we committed more time to the second example submissions because we felt that working on the function case was a richer and more relevant problem to solve. So, for future work, we hope to integrate Jay's function wrapper code into our own code so that it works for any type of question. This way, the instructor is not limited to questions that ask the final answer to be solely in the form of a function, eliminating the need to constantly modify code.

For a more technical improvement, we do have some issues with how we flag code that does not run. Our flag may be flawed if a student's code is able to run on some test cases but not on others. For a simple example:

Let's say a student's code does not run for test case 1, they receive a badness of -1 under our code. For test case 2 they have a single error with badness 1. When we sum the badness of the test cases with equal proportion, the student is given a cumulative badness of 0. Our code would then assign the student a grade of 100 even though their code didn't run for a test case. This is an issue that we did not see in our example but it could feasibly happen. To prevent this we could introduce a completely unique flag for DNR. An addition to the code would be to try to assign grades when a student's code DNR's in some cases but does not DNR in others. Due to time constraints, we created a solution assuming that this case does not happen. For safety, a professor could look at the case-dependent badness and if there are negative and positive badness for a single student, this would be cause for concern.

These are some niche issues, we provide some more broad commentary below.

What our code does well:

- Works perfectly with functions as submissions
- Checks for timeout in students' code to skip over code that takes too long to run
- Gives the user complete control over what to do with the errors made throughout the class size but also the option to trust the defaults as being reasonable enough
- Takes care of giving user the flexibility to use the code the way they like with options to:
  - customs parameter in *checkEquiv* which pass custom functions to test against the student's answer
  - errCodes parameter in *checkEquiv* with error codes for when test in customs fail
  - U parameter in *scoresFunc* lets the user enter a vector of the errors to test over all students' list of errors
  - WErr parameter in *scoresFunc* lets the user specify custom weights of each error

  - WCase parameter in scoresFunc lets the user specify custom weights of each case
  - curve parameter in *gradeFunctionsSimple* lets the user decide if they want to curve the grades to a normal distribution
- Also has a set of reasonable defaults so the user can use the code with minimal efforts;
  - U parameter in *scoresFunc* defaults to construct a list of errors by taking unique errors from the n x m matrix with each students errors
  - WErr parameter in *scoresFunc* defaults to giving every error the same "badness"
  - WCase parameter in *scoresFunc* defaults to giving every test case the same "badness"
  - curve parameter in *gradeFunctionsSimple* defaults to not curving the grades

What are our code's drawbacks:

- The student submissions have to be functions to work with our code
- The badness flag for DNR could interfere with regular badness
- The wrapper function *gradesFunctionsSimple* uses all defaults, some more customization would be nice
- The conversion from badness to grades is difficult and we could provide the user more control over that process
- Our methods don't account for random values in code, but we could allow a test for the code which is effectively a confidence band around the professor's answer.