grade Package Tutorial

Leif Johnson

February 20, 2009

1 Purpose and Motivation

1.1 Motivation

This package came about because of work I was doing with Moodle `http://www.moodle.org`. Specifically I made a Moodle question type plugin that allows *R* to be integrated directly into Moodle questions. In brief, the Moodle server sends the question to a remote server for processing. This server does stuff to the question, and can process chunks of it however it pleases. I have a server doing this with *R*.

Each Moodle question has a collection of answers with associated feedback and grades. When a student submits an answer to a Moodle question, Moodle iterates through the answers to find a match. The student then gets the appropriate feedback and grade for that question. The Moodle question type can grade single value numerical answers, but it can't do anything more complicated than that.

My solution was to build in a *remote grading* option into the question type. If *remote grading* is enabled, the Moodle server sends the student's answer back to the remote server for grading. This raises issues of text processing and 'answer' matching. This is what the *grade* package provides.

1.2 The Problem

We want to be able to grade questions from within the question type, with minimal amount of work from the question writer. A simple example of a question would be:

Question: Given that $< data >$ follows a normal distribution with standard deviation $< sdev >$, find a 95% confidence interval for the mean.

answer: $grade.interval(x.bar + s.dev * qnorm(0.975) * c(-1, 1), studentans)$

So the remote server needs to evaluate each possible answer and return a binary true/false decision back to Moodle. This involves calculating the correct answer, converting the student's answer from *text* to the appropriate form, and deciding if they match. *grade* provides functions to do this.

2 Usage

Usage is intended to be very simple, and hopefully it is. Each grading function follows the same basic format:

$$grade.function(correctans, studentans)$$

The grading function returns *TRUE* if *studentans* is correct, and *FALSE* otherwise.
correctans is the 'correct' answer. *studentans* is the student answer. Although you are not required to use them in that order, it is a good idea. Functions typically enforce stricter requirements on *correctans*. E.g. *grade.interval* requires that *correctans* have length 2 and errors if it does not. *grade.interval* does not error if *studentans* does not have length 2, though the answer will be incorrect.

There are 5 options in common as well:

- *tolerance* Either a numeric or a string representing a numeric (NOT INF or NA). Usage varies by grading function, but is typically a component wise tolerance.
- *useeval TRUE/FALSE*. Defaults to *TRUE*. If *TRUE* text elements are evaluated using *eval*. If *FALSE* text elements are coerced with *as.numeric*.
- *usena TRUE/FALSE*. Defaults to *FALSE*. If *TRUE*, *NA* is an accepted value, if *FALSE NA* is not considered to be valid.
- *useinf TRUE/FALSE*. Defaults to *FALSE*. If *TRUE*, *Inf* and *-Inf* are accepted values. If *FALSE*, *Inf* and *-Inf* are not considered to be valid.
- *quiet TRUE/FALSE*. Defaults to *TRUE*. If *TRUE*, functions output as little extra information as possible. If *FALSE* there are more warning messages. Can be use full for debugging or tracing failures.

2.1 Simple Examples

First some intervals. Note the usage of 'usena', and 'useinf'.

```
> library(grade)
> x <- c(1, 2)
> grade.interval(x, "[1,2]")

[1] TRUE

> grade.interval(x, "[1,2.02]")

[1] FALSE

> grade.interval(x, "[1,2.02]", tolerance = 0.03)

[1] TRUE

> grade.interval("[NA, 1]", c(NA, 1), usena = T)

[1] FALSE

> grade.interval(c(1, Inf), c(1, Inf))

[1] FALSE

> grade.interval(c(1, Inf), c(1, Inf), useinf = T)

[1] TRUE
```

To grade sets there are *grade.set* and *grade.orderedset*, the only difference is that *grade.orderedset* requires the sets to be in the same order.

```
> set1 <- "[1,2,3,5]"
> set2 <- "[5,3,2,1]"
> grade.orderedset(set1, set2)

[1] FALSE

> grade.set(set1, set2)

[1] TRUE
```

```
> set3 <- c(NA, Inf, pi)
> grade.orderedset(set3, set3)

[1] FALSE

> grade.orderedset(set3, set3, usena = T, useinf = T)

[1] TRUE
```

It can be used to verify that a student's answer could be a discrete probability dist.

```
> grade.discreteprobability(NULL, c(0, 1/2, 1/2, 0), checkcorrect = FALSE)

[1] TRUE

> grade.discreteprobability(NULL, c(-1, 1/2, 1/2, 1), checkcorrect = FALSE)

[1] FALSE
```

And to check against a target distribution, order is enforced if *ordered=TRUE*, and not enforced if *ordered=FALSE*:

```
> correct <- c(0, 1/2, 1/4, 1/4)
> sans <- "[0, 1/4, 1/4, 1/2]"
> grade.discreteprobability(correct, sans)

[1] TRUE

> grade.discreteprobability(correct, sans, ordered = T)

[1] FALSE
```

Lastly, we have a function to check if a number is negative, and one to compare a single value.

```
> grade.negative(NULL, -1)

[1] TRUE

> grade.negative(NULL, -Inf)

[1] FALSE
```

```
> grade.negative(NULL, -Inf, useinf = T)

[1] TRUE

> grade.number(1, 1)

[1] TRUE

> grade.number(3.141, "pi", tolerance = 0.002)

[1] TRUE
```

2.2 Parsing Input

There are 4 functions involved in parsing input, *grade.isscalar*, *grade.parse*, *grade.parsechunk* and *grade.parseset*. Generally, you will just want to use *grade.parse*. These functions return a vector of values if all checks are passed, *NULL* otherwise.

```
> grade.parse("[1, 2, 3]")

[1] 1 2 3

> grade.parse("NA")

NULL

> grade.parse("NA", usena = T)

[1] NA

> grade.parse("[-Inf, Inf]")

NULL

> grade.parse("[-Inf, Inf]", useinf = T)

[1] -Inf  Inf
```

Note that you can pass R objects in as well. These need to conform to whatever requirements you pass to the parse function:

```
> grade.parse(c(1, 2, 3))
```

```
[1] 1 2 3
```

```
> grade.parse(NA)
```

```
NULL
```

```
> grade.parse(c(-Inf, Inf), useinf = T)
```

```
[1] -Inf  Inf
```

Checks are made with the *grade.isscalar* function:

```
> grade.isscalar(c(1, 2))
```

```
[1] FALSE
```

```
> grade.isscalar(Inf, useinf = T)
```

```
[1] TRUE
```

2.3 eval vs as.numeric

Using *eval* to parse input that is coming from random sources can be a problem, but the grade.parse functions **will not eval text containing parenthesis or assignment operators**. It is unlikely that anything malicious can slip through.

```
> x <- NULL
> grade.parse("x <- 1")
```

```
NULL
```

```
> x
```

```
NULL
```

```
> grade.parse("rm(list=ls())")
```

```
NULL
```

```
> ls()
```

```
[1] "correct" "sans"    "set1"    "set2"    "set3"    "x"
```

```
> eval(parse(text = "rm(list=ls())"))
> ls()

character(0)
```

Using *eval* should generally be safe. It does have a significant advantage over *as.numeric*:

```
> grade.parse("1")

[1] 1

> grade.parse("1", useeval = F)

[1] 1

> grade.parse("pi")

[1] 3.141593

> grade.parse("pi", useeval = F)

NULL

> grade.parse("1/2")

[1] 0.5

> grade.parse("1/2", useeval = F)

NULL
```

I recommend that you use *eval*, which is why it's the default.