At Ulm University, we currently use [RTutor](http://skranz.github.io/RTutor/) for several elective courses where students solve interactive problem sets at home. They can test their solutions and get automatic hints, and then submit their solution. Starting next year, we plan to use RTutor in a new compulsory data science project course in [our business and economics bachelor](https://www.uni-ulm.de/mawi/bachelor-wiwi). For a compulsory course, it seems even more important that the problem sets are well polished to avoid excessive frustration by students who might get stuck. In this blog post, I want to illustrate how the new companion package [RTutorSAGI](https://github.com/skranz/RTutorSAGI) can be used to systematically analyse students’ submission in order to improve the RTutor problem sets used in class.

Also note that there have been substantial updates for RTutor itself in July 2019. There is finally a [pkgdown](https://pkgdown.r-lib.org/) powered project page:

<http://skranz.github.io/RTutor/>

In the [Manuals](http://skranz.github.io/RTutor/articles/) section you can find the new, fully rewritten documentation. Other new features are easier customization of hints and improved adaptive automatic hints (see [here](http://skranz.github.io/RTutor/articles/03_HintsAndTests.html)).

Now, let us directly exemplify [RTutorSAGI](https://github.com/skranz/RTutorSAGI) (SAGI stands for Submission Analysis for Grading and Improvement) with some code:

library(RTutorSAGI)

# Load all submission files from directory ./sub

sub.li = load.subs(sub.dir="sub", warn=FALSE)

# Create summary of solution attempts by chunk

sum.df = analyse.subs(sub.li, rps.dir="org\_ps",

just.summary = TRUE, protracted.minutes = 30)

# Show top 10 of chunks were students

# needed most solution attempts

sum.df %>%

arrange(-err) %>%

head(10)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ps.name** | **chunk** | **chunk.name** | **solved** | **unsolved** | **protracted** | **err** | **users.err** | **hint** | **users.hint** |
| ps\_1b1\_ols | 22 | 5\_a | 14 | 0 | 10 | 127 | 10 | 28 | 7 |
| ps\_1d | 3 | 1\_b\_2 | 15 | 0 | 8 | 109 | 8 | 18 | 7 |
| ps\_2a\_logit | 18 | 4\_e\_2 | 15 | 0 | 4 | 98 | 4 | 15 | 3 |
| ps\_2a\_logit | 2 | 1\_b | 15 | 0 | 9 | 88 | 9 | 65 | 8 |
| Intro | 13 | 3\_b | 15 | 0 | 13 | 88 | 13 | 11 | 8 |
| ps\_1b2\_iv | 23 | 4\_2\_f | 14 | 0 | 9 | 83 | 9 | 46 | 8 |
| ps\_2a\_logit | 20 | 5\_1\_b | 15 | 0 | 10 | 82 | 10 | 62 | 10 |
| ps\_1b1\_ols | 12 | 2\_d | 15 | 0 | 6 | 80 | 6 | 16 | 5 |
| ps\_2a\_logit | 11 | 3\_a | 15 | 0 | 9 | 74 | 9 | 25 | 8 |
| ps\_2a\_logit | 10 | 2\_f | 15 | 0 | 7 | 70 | 7 | 11 | 5 |

The code analyses all students’ submissions of the 8 RTutor problem sets of one of my master level economics courses. The created data frame sum.df provides information for the different chunks of how hard they were to solve. From the columns solved and unsolved we learn that in the end all students who tried mastered to solve all 10 shown chunks (and this also holds true for the non shown chunks.). This is reassuring but it was also a small class. The column err shows how often in total students checked their solution for the corresponding chunk and the correctness test failed. This was most often the case for the chunk in the first line, i.e. chunk 5\_a in my problem set ps\_1b1\_ols: 127 times. We also see that 10 students (column users.err) entered at least one wrong solution. For this chunk, a hint was requested in total 28 times and 7 users asked at least once for a hint.

The column protracted is an indicator whether or not it took a long time to finally solve the chunk. More precisely, we compute the duration between the first time a student entered a wrong solution (that differed from the original shown task code) and the time when that student finally solved the chunk. We then count the number of students for whom that duration is longer than 30 minutes. We set the threshold 30 minutes via the argument protracted.minutes in the call to analyze.subs. Here 10 from 14 students took longer than 30 minutes. That does not neccessarily mean that they worked more than 30 minutes on the chunk. Many of the students could have decided to take a break because they could not solve it immediately. We get more detailed information by looking at the individual chunk logs illustrated further below.

Given that students in this class had a total of 164 tasks in RTutor problem sets where they needed to enter code, I decided for the benefit of future students to smooth some problematic tasks by changing the structure of some problem sets and adding more informative custom hints.

Let us look at the chunk 5\_a discussed above. In this task students should create the following function that will later be used for a systematic Monte-Carlo Simulation of an OLS estimator:

sim.and.est <- function(beta0=100,beta1=-1,...) {

data = sim.data(beta0=beta0, beta1=beta1, ...)

# Compute the estimated coefficient beta.hat from a linear regression

# of the demand function

beta.hat = coef(lm.fit(y=data$q, x=cbind(1,data$p)))

# Generate and return a data.frame containing

# the estimated and true coefficients

ret = data.frame(coef.name=c("beta0","beta1"),

est.coef=beta.hat, true.coef=c(beta0,beta1))

return(ret)

}

Writing a function tends to be tricky for inexperienced users. Even though I gave students some additional information and a partial solution, it is ex-post understandable that several students got stuck here.

For improving a problem set, it is helpful to know how students tried to solve the task and what exactly went wrong. For this purpose, we can run once the RTutorSAGI function write.chunk.logs:

write.chunk.logs(sub.li,logs.dir = "chunk\_logs", rps.dir = "org\_ps")

The call above saves in the directory ./chunk\_logs for every chunk where at least one user entered a wrong solution an R file that contains a log of the solution attempts of all users. Here is an exercept from the log of chunk 5\_a:

# ps\_1b1\_ols 5\_a

# 14 users solved (10 took long time) and 0 failed.

# Failed checks: 127, Hints: 28

# NEW USER solved after 3 failures (3.9 mins) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# A function that simulates random prices and demand

# and estimates beta.hat

sim.and.est = function(beta0=100,beta1=-1,...) {

data = sim.data(beta0=beta0, beta1=beta1, ...)

# Compute the estimated coefficient beta.hat from a linear regression

# of the demand function (use lm.fit or lm and coef)

reg<-lm(q~p)

coef(reg)

# enter your code here ...

# Generate and return a data.frame containing the

# estimated and true coefficients

ret = data.frame(coef.name=c("beta0","beta1"),

est.coef=beta.hat, true.coef=c(beta0,beta1))

return(ret)

}

# \*\*\* 102 secs later...

# A function that simulates random prices and demand and estimates beta.hat

sim.and.est = function(beta0=100,beta1=-1,...) {

data = sim.data(beta0=beta0, beta1=beta1, ...)

# Compute the estimated coefficient beta.hat from a linear regression

# of the demand function (use lm.fit or lm and coef)

reg<-lm.fit(cbind(1,p), q)

coef(reg)

# enter your code here ...

# Generate and return a data.frame containing the

# estimated and true coefficients

ret = data.frame(coef.name=c("beta0","beta1"),

est.coef=beta.hat, true.coef=c(beta0,beta1))

return(ret)

}

# \*\*\* 24 secs later...

# A function that simulates random prices and demand and estimates beta.hat

sim.and.est = function(beta0=100,beta1=-1,...) {

data = sim.data(beta0=beta0, beta1=beta1, ...)

# Compute the estimated coefficient beta.hat from a linear regression

# of the demand function (use lm.fit or lm and coef)

reg<-lm.fit(p, q)

coef(reg)

# enter your code here ...

# Generate and return a data.frame containing the

# estimated and true coefficients

ret = data.frame(coef.name=c("beta0","beta1"),

est.coef=beta.hat, true.coef=c(beta0,beta1))

return(ret)

}

# \*\*\* 9 secs later... asked for hint.

# \*\*\* 100 secs later... solved succesfully!

# NEW USER solved after 4 failures (4.6 hours) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# ... over 2000 more lines ...

The first user makes several mistakes: not using the variables p and q from the data frame data and not assigning coef(reg) to beta.hat. (These errors were not so easily detected by the user because in earlier tasks the variables p, q and beta.hat were assigned, i.e. the function did run without error in RStudio). None of the 3 first attempts corrected that mistake. In the new RTutor version the automatic hint provides some information about the problem:

Warning: Your function uses the global variable(s)

beta.hat, p, q

Often global variables in a function indicate a bug and you just have

forgotten to assign values to beta.hat, p, q inside your function.

Either correct your function or make sure that you truely want to

use these global variables inside your function.

However the automatic hints in RTutor don’t allow to analyse step by step the different commands inside a function. Looking at more users in the log, one finds that there are many ways to write an erroneous function. This means custom hints were also of limited use here.

In the revision of my problem sets, I made the following change. If students shall write a function, it is now almost always done in a two step procedure. In an initial task students just write the future function body as normal R code for some example values of the function arguments. Since the code is not yet in a function, RTutor can nicely test it step by step and provide informative automatic hints. In a subsequent task students have to copy that function body into a function where I already provide the function header and closing } in a fill\_in block.

For another example, let us look at a much simpler task from the same problem set:

Using the command cbind, generate the matrix X of explanatory variable whose first column consists of 1 and the second column consists of p

The sample solution is just:

X = cbind(1,p)

Here is an extract from the corresponding chunk log that was created by write.chunk.logs:

# ps\_1b1\_ols 1\_b\_2

# 15 users solved (4 took long time) and 0 users failed.

# Failed checks: 8 Hints: 4

# NEW USER solved after 3 failures (3.1 mins) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

cbind(1,p)

# \*\*\* 29 secs later...

x <- cbind(1,p)

# \*\*\* 13 secs later...

x = cbind(1,p)

# \*\*\* 98 secs later... asked for hint.

# \*\*\* 49 secs later... solved succesfully!

# NEW USER solved after 1 failures (6.7 mins) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# asked for hint.

# \*\*\* 359 secs later...

x= cbind(1,p)

# \*\*\* 41 secs later... solved succesfully!

# NEW USER solved after 2 failures (2.7 mins) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

x=cbind(rep(1,T),p)

# \*\*\* 22 secs later...

x=cbind(rep(1,T),p)

# \*\*\* 103 secs later... asked for hint.

# \*\*\* 38 secs later... solved succesfully!

Here is not really a need to customize any hint because every student finally managed to solve the chunk in reasonable time. Nevertheless, it is a nice example to showcase a simple adaptive custom hint. In the new version of the problem set solution file, this chunk is now specified as follows:

X = cbind(1,p)

#< add\_to\_hint

if (exists("x") & !exists("X")) {

cat("It looks like you assigned the value to 'x' (lowercase),

but you shall assign the value to 'X' (uppercase).")

}

#>

Even more than customizing the problem sets of my course, I used the insights from the analysis of submissions to to improve the automatic hints and tests of the RTutor package. For example, it is quite valuable to teach students how to effectively use longer dplyr chains. This can be done now in RTutor problem sets in most cases without the need for any customized hints. Assume you just specified in your solution file the following command

dat %>%

group\_by(a) %>%

summarize(mean\_x = mean(x)) %>%

arrange(-mean\_x)

and the student enters the following wrong code

dat %>%

group\_by(a) %>%

summarize(mean\_x = sum(x)) %>%

arrange(-mean\_x)

Then RTutor’s automatic hint gives the student a relatively detailed error analysis:

In your following pipe chain, I detect an error in the 3rd element:

dat %>%

group\_by(a) %>%

summarize(mean\_x = sum(x)) %>% !! WRONG !!

arrange(-mean\_x)

It is correct to call summarize. But: Your argument mean\_x = sum(x)

differs from my solution, where I call the function mean.

