An RTutor Problem Set as Bachelor or Master Thesis – A Guide

Sebastian Kranz, Ulm University

Version from 2015-10-19

Contents

1 Part 1: Overview of the main steps

This guide helps you writing an RTutor problem set based on an economic article as part of your Bachelor or Master thesis. Such a thesis requires substantial time and work: you need to dig deep into the topic of the article, into econometric methods and into R programming. So this is not the easiest way for writing a thesis, but from those students who have already mastered this challenge, I got a lot of positive feedback: You learn a lot, and it can be quite fun and rewarding to create and share a nice RTutor problem set.

1.1 Step 1: Find an interesting economic article

First you need to find an interesting article for which the data (and ideally also the original code) is accessible.

New: A list of selected articles with data from the *Review of Economics and Statistics* can be found here:

http://econfin.de/rtutor/restat.html

(I have sorted the beginning of the list according to which articles I found more interesting.)

Article database: I also have created a searchable database of articles with data from selected journals (AER, AEJ, REStud). It can be found here:

http://econfin.de:3838/articles/

You can search for specific topics in the database by entering different Journal of Economic Literature (JEL) codes, that classify economic articles by different topics.

The shown articles contain a link to the journal website of the article (example). You can download the article there (due to access restrictions this works only from the university network).

1.1.1 Zip file with data and code

From the articles website you can typically also download a zip file that contains the articles' data sets and codes. If you download and open the zip file, you typically find a README file (possibly in pdf or txt format). It describes the contributed data sets and codes. In particular, it is noted whether all data for replication is available or whether some data must be bought or is inaccessible. You should choose an article where all data is available, or at least most parts of the article can be replicated with the available data.

1.1.2 Contributed code

Most articles have contributed Stata code in `.do` files, which can be opened with any text editor. Sometimes there is also code in other languages like Matlab `.m` or C, Fortran, Python, R, ... As a rough rule of thumb, articles that only contain Stata code are easier to replicate than articles that also contain Matlab or other code. If you search for articles with JEL code, you see for many articles already an overview of the types and size of code files, e.g.

```
Inventories, Lumpy Trade, and Large Devaluations (55.41 MB, aer, 2010/12)
...
Code in KB: do: 53 m: 449; Data (decompressed): 206.6 MB
```

This article has 54 KB of Stata code (in .do files) and 449 KB of Matlab code in (.m) files and around 206.6 MB of data files.

1.1.3 Selecting an article

Best find some candidate articles and then talk with me before selecting one. Some constraints have to be satisfied, e.g. you should not take some article for which already a RTutor problem set has been created.

1.2 Step 2: Take a look at an existing interactive problem set

(Note: You can swap steps 2 or 3) On the Github page of RTutor https://github.com/skranz/RTutor you find links to several RTutor problem sets that have been created as part of a Bachelor- or Master thesis. Try installing one or more problem sets from their Github page and try them out. This helps you to get an idea, how problems could be structured, which elements you like, and which you like less. Also you can learn a bit more R. For starts, I would recommend:

https://github.com/JonasSend/RTutorTopIncomeTaxation

You can also download the whole Github package as ZIP file and take a look at the source code of the problem set. If the problem set is called `myps`, the source code of the problem set is in the file `myps_sol.rmd` in the folder `/inst/ps/myps/`.

1.3 Step 3: Work through the article and replicate the code in R

Now work through your article and the code. Then try to replicate most results in R. At this point you don't yet have to think about the design of the problem set, just try to make the analysis run in R. It is a good idea, however, to already use helpful R packages like `dplyr`, `ggplot2`, `reshape2`, `tidyr` or `lfe` to write more elegant and shorter R code. Take a look at the section of recommended packages below! You can learn more about useful functions in these packages, e.g. by solving an existing problem set in Step 2.

If you cannot replicate all results in R, or you find bugs in the Stata code, write down all these points. This will be a useful appendix to your thesis.

Even though your final "product" will be written in R, it can be helpful to run the given code in Stata, in order to better understand it. I can send you a guide for several ways how to use Stata at Ulm University.

1.4 Step 4: Create the interactive RTutor problem set

Now you can start creating the RTutor problem set. An overview of creating problem sets is given here:

http://econfin.de/rtutor] (link to pdf)

https://Github.com/skranz/RTutor/blob/master/vignettes/Guide_for_Developing_Interactive_R_Problemsets.md (markdown)

As an initial template for your problem, either use the example file in the RTutor package:

https://raw.githubusercontent.com/skranz/RTutor/master/inst/examples/Example_sol.Rmd

or a solution file from one of the existing problem sets. First try to make the example run before you build your own exercises. Note that some bugs and features are not yet well documented. So just drop by, if it does not run.

Part 2 below gives detailed advice on how the problem set should be structured, and which R packages seem useful. Also drop by regularly to get feedback and discuss problems.

1.5 Step 5: Ask your advisor soon for feedback and improve the problem set

Try to have ready a first version working version of your interactive problem set quite soon and ask your advisor for feedback. This version must not yet contain all exercises, but one should be able to try it out.

You should have at least half of the time for your thesis left, when you send a first version of your problem set to your advisor.

You need the second half of time to improve the problem set.

From my experience, the first working drafts of a problem set, may already have many good ideas, but are still very far away from a nicely polished final product. Most problem sets have been substantially restructured and improved compared to the first version. Typically there will be quite some iterations, where you send a new version of the problem set to your advisor, discuss it together and then improve the problem set. This gradual improvement process takes substantial time, however.

The main reason why a thesis falls below the desired quality is that you send a draft to late to your supervisor and don't have enough time for the multiple rounds of feedback and improvement.

1.6 Step 6: Host your problem set on Github and shinyapps.io

If you have finished a first or later version of your problem set, you can already put it on the web (you can always update the web versions). I would recommend to host it as an R package on Github. How to do it, is explained in detail here:

Deploy RTutor problem sets on Github

Furthermore, I would recommend to also host your problem set on shinyapps.io. This allows users to directly solve your problem set in the web without having to install R. (The free service is restricted to 30 access hours per month, however). This is described here:

Deploy RTutor problem sets on shinyapps.io

1.7 Step 7: Write your thesis

Your problem set will be the core and main part of your thesis. Many essential parts a thesis, like discussion of related literature, references, own discussion etc., can also be part of your problem set. (In the problem set, you may want to put some longer literature discussion in a footnote). Still you may add some extra aspects in your written thesis, e.g. discuss how you have designed your problem set and which problems you have encountered.

Many existing problem sets also contain their thesis in the Github repositories, which can give you some idea, how you can create the thesis from your problem set.

You can write your thesis using Word, Latex, Latex via Lyx, or directly in a Markdown format. When you create your interactive problem set with the command `create.ps(...)` a file with the ending `..._output_solution.Rmd` is created in your working directory. It contains a RMarkdown version of your solved problem set, which you can translate from RStudio to Latex or Word in order to easier include your problem set in your thesis.

To create a Latex file add the following codes as first lines of the `..._output_solution.Rmd`

```
---
output:
  pdf_document:
    keep_tex: true
---
```

If you want to convert to Word, add instead:

```
---
output:
  word_document: default
---
```

2 Part 2: Suggestions for your RTutor problem set

This is a list of suggestions when you create as part of your Bachelor or Master thesis an interactive problem set that reproduce the analysis of a published economic article. Also take a look at the general RTutor Guide for making problem sets.

2.1 Main Philosophy of your problem set

```
Send the user on an interesting, data-driven journey into the economic contents of the article
```

The problem set shall resemble an interactive article. It shall *not* be a tough problem set from a course that is designed to make students learn by figuring out much stuff for themselves. The user shall enter some lines of code, but she shall not forced to work too long on solving it. So don't worry that the problem set is too simple or too little work for the user.

2.2 Target user

Design most parts of your problem set with the following typical user in mind:

1. The user is somebody who finds your problem set on the internet, is interested in the topic and wants to try it out.
2. The user knows already a little bit of R, but does not necessarily know how to use packages like `dplyr`.
3. The user is primarily interested in the economic analysis of the article.
4. Whenever the user has computed a summary statistic, has run a regression or has shown a plot, she wants to read a short economic interpretation of the results. Try to always add text with such interpretations: mostly a good place is directly below the code chunk where the user shall perform the computation.
5. While the user is also interested in learning some R tricks and seeing and adapting examples of well written R code, she does not like spending time searching through the internet or help files in order to find R commands or their precise syntax. She rather wants you to first give an example, that she can adapt. She also strongly prefers short, elegant R code & tricks from existing packages. She does not like inelegant, complicated code.
6. While the user likes solving and reading a well designed problem set, she also can quickly become bored. If solving the task becomes too complicated or she must do too much repeated work, she will just stop solving the problem set. E.g. if many similar computations have to be performed, just let the user solve one or two and provide the remaining computations in the task.
7. If the user has good knowledge of R and economics, she should be able to solve the problem set quite quickly by adapting the examples you provide in the task description.

2.3 Tips for designing yor problem set

2.3.1 Start quickly with an interesting data set.

Give only a brief background of the content and try to start quickly with loading an interesting data set. Let the user generate summary statistics and figures of interesting aspects.

2.3.2 Tell the economic story with the data

Typically, show first the data and then based on the data, develop step by step the story, hypothesis and results of the paper. Nevertheless, you still should give a very short summary, the length of a an abstract, at the beginning of the problem set.

2.3.3 Pick good variable names and good variable descriptions

An important part of your initial work will be the generation of a variable description text file. It has the following format like in the following example (Header & 2 entries):

```
orgvar | var | descr

minority_dummy | minority | Dummy variable 1 if the account holder belongs to the minority community

ins_adj | above_insured | Dummy variable; 1 if the account holders total balances are above the deposit
```

The column `orgvar` is the name of the variable as given in the original data. The column `var` can be an alternative name that is easier to read. Separate words with `_`. Finally, the column `descr` provides a description of the variable, typically one or two sentences.

When calling `create.ps`, provide the file name of this file as argument `var.txt.file`. The variable description will then be shown in the data explorer of the HTML problem sets. Furthermore, the variable description will by default be shown as a tooltip on the column name when you display a data.frame in some code chunk.

You can translate the original variable names to your new variables in a code chunk of your problem set by using the function `translate.var.names` which is part of RTutor. Example:

```
# Read original Stata data
term = read.dta("term_deposit_data_file1.dta")
# Automatically translate the variable names
term = translate.var.names(term)
```

2.3.4 Design the problem set for HTML

The new RTutor allows two sort of problem sets: HTML-Shiny-based or Markdown-RStudio-based. Even though for longer courses the Markdown-RStudio-based approach has some advantages, with our target user in mind, you should definitely design the problem set for the HTML-Shiny-based format.

2.3.5 Start with an overview and table of content and end with a conclusion and bibliography

Your first 'Exercise' can have the name 'Overview' and just briefly describe the structure of your problem set and give a table of content that briefly describes the exercises. Here the user should not solve anything here.

Your last 'Exercise' can have the name 'Summary' or 'Conclusion' and can contain some final thoughts and bibliography that lists all used references.

You can create these two special exercises after you have created the 'core' of your problem set.

2.3.6 Allow exercises to be solved independent from each other

RTutor has the option to use results from an earlier exercise in a later exercise. This requires that the user can only start the later exercise if he has finished the earlier exercise. Try to avoid this structure in your problem set. Instead allow each exercise to be solved separately. This means that typically you will load some data at the beginning of each exercise, even though you have already loaded the data before.

2.3.7 Split long exercises

Try to avoid very long exercises. Better split them up in sub-exercises, e.g. Exercise 3.1 , Exercise 3.2 and so on.

2.3.8 Use info blocks for background information and variable descriptions

When designing problem sets, you often face the following conflict. On the one hand, you may want to give detailed background information. On the other hand, you want your tasks sufficiently short, as you don't want to force the reader to read a lot before she can start analyzing the data. To resolve this conflict, you can put a lot of information into info blocks. The user can view info blocks whenever she likes. You should generally put a description of the relevant variables of the current data set in info blocks.

There is also the possibility to use footnotes that contain code chunks that the user can optionally solve. This feature, has first been used in the problem set `creditboomsgonebust` by Thomas Clausing, but is not yet well documented. Ask me if you want to use these footnotes.

2.3.9 Create short code chunks and frequently add explanations and discussions afterwards

Try to avoid too many commands in one code chunk, but rather create more chunks. It is totally OK if the user only has to enter one or two commands in each code chunk. Whenever the user performed some analysis, provide a short explanation or discussion afterwards that helps the user to interpret the results.

2.3.10 Interpret significant coefficients and effect sizes

After showing regression results, try to verbally interpret some economically interesting significant coefficients. What do they mean (one unit increase of x changes y by ??? units)? Do they seem economically substantial or rather not so? How big is the uncertainty mirrored in the standard errors? You can also ask the user for the quantitative interpretation via a quiz (see below).

Also consider using the function `effectplot` in my package `regtools`. It helps to compare the effects that changes in different explanatory variables have on the dependent variable in a standardized fashion.

If coefficients are not significant, you may be more careful in interpreting them. Many researchers have the opinion that you should not interpret insignificant coefficients quantitatively.

2.3.11 Only load the data you currently need

Don't load and prepare all data sets at once, but try to start small and only load the data set you need for the current steps of the analysis.

2.3.12 Start with a descriptive or graphical analysis before you prepare the data

It is OK if at some later point you want to teach the user how to modify and prepare data for the analysis (such skills are definitely useful since much time is usually spent with data preparation). Yet, try to start the problem set with loading data and showing some description or plot. Relegate data preparation to later parts. This means you may provide the user with a data set that you have prepared already yourself. In a later exercise you can explain how you have prepared that data.

2.3.13 Don't let the user repeat too often boring tasks

While you can ask the user some time to prepare or modify data, don't let him do all data preparation. Let him learn interesting and useful things, but don't let him do boring work.

2.3.14 Write elegant R code and use the recommended packages

There are many ways to perform some analysis in R, but one goal of the problem sets is to teach rather elegant solutions. "Elegant" typically means just a few lines of code that can be easily read. Of course, it is not so easy to write elegant code when you are yourself not yet an R expert. One thing you definitely should do is to read the section *"Recommended Packages"* and use those packages for typical tasks like data manipulation, plotting or showing regression outputs in a nice format.

2.3.15 include auxialliary functions in an `extra.code.file`

You may want to write some own functions that simplify some steps of the analysis and make the user code shorter. You can put them into an extra .r file whose file name you pass as argument `extra.code.file` to `create.ps`. Your functions can then be called everywhere in the problem set.

2.3.16 Use quizzes

You can also ask the user a question about the data or regression results in form of a quiz. There is now an addon that allows to include multiple choice and other quizes in a nice format in shiny based problem sets. It is not yet well documented, but the file `QuizExample_sol.Rmd` in the folder `/inst/examples` contains an example, how this quiz addon can be used. The examples, are hopefully self explanatory. Important is that you set the parameter `addons="quiz"` in your call to `create.ps`.

Alternatively, you can include quizes as part of a normal R code chunk. Here is an example, how you can implement a quiz chunk:

```
c) Look at the different plots to answer the following questions (replace '???' with the right answe
```{r "2.1 c)"}
#< task_notest
# Do you see a particular effect in open-end fund starts and net flows during Regime 2 compared to t
# sol31 = "???" Assign "yes" or "no" to sol31 and remove the comment
#>
sol31 = "no"
```
```

2.3.17 Make life easy for the user: give code examples

If the user shall write some code that is a bit more complicated, try to give an example in a task plot first that he can adapt. Remember that the target user does not want to figure out complicated commands on his own but wants to see quick results.

2.3.18 Make life easy for the user: give part of the code

If there is a complicated command or task, you can also give part of the code to the user in the problem set and make him adapt the code. Here is an example, how you can construct that in your solution file:

```
```{r "2.2.1 c)"}
#< task_notest
# Replace the ??? in the code below and uncomment the command.
# plot(???, ??? , ylim = c(0,6.5))
#>
plot(x = open$nfo_open_date, y = open$entry_load_nfo, ylim = c(0,6.5))
```
```

Of course, once the user has seen the code. You could ask him to write the full plot command next time.

2.3.19 Use compute blocks if you ask for computations in several steps

2.3.20 Not yet implemented

Maybe at some point you want to ask a more complex computation from the user instead of just a single line of code. In this case consider using a compute block that is designed to give sensible hints for students.

2.3.21 If you have a hard task put it in an own exercise and / or be generous with your hints

Even if overall the goal is that the problem set is not too hard to solve. You may want to add one or two more tricky exercises. Then make sure that you put them in an extra exercise, which can be solved optionally. Otherwise be very generous with the hint, e.g. provide the solution in your hint.

2.3.22 Except you have a very good reason, don't let the user write unelegant, complicated code if you know a nicer way.

Sometimes you think it could be a good idea that the user first uses a complicated way to solve problem and you only later tell how she can solve it more elegantly. Typically this not such a good idea and it is better to skip the inelegant solution and immediately start with the elegant solution.

2.4 Recommended packages and functions

This sections gives recommendations for packages and functions you should use for common tasks in your problem sets.

2.4.1 Regressions

2.4.1.1 present regression results: stargazer

The package `stargazer` contains the function `stargazer` that allows to show regression results as nice HTML tables similar to the tables shown in scientific articles. The package `texreg` contains similar functions, but seem a bit less powerful and not compatible with as large a set of regression models.

That function `showreg` in my package `regtools` is a wrapper to `texreg` or `stagazer` and also allows to show robust standard errors (not clustered ones) in a simple fashion (see below).

2.4.1.2 basic regression functions

The base R functions `lm` and `glm` allow linear regressions and generalized linear models, including Tobit and probit regressions. But there are many specialized functions and packages for other sorts of regressions, e.g.

- `lfe`: for estimating large panel data sets with fixed effects, also allows IV estimation.
- `mlogit`: discrete choice data

For other models search the internet.

2.4.2 Robust standard errors: lfe, sandwich, multiwayvcov

Dealing with robust standard errors in R is traditionally a bit more complicated than in Stata, but can be done.

Often papers report 'cluster robust' standard errors. To obtain these, it is best to run your linear regression or IV regression with the function `felm` in the package `lfe`, which allows for clusters in the regression formula (see the documentation for `felm`). If you output the results with `stargazer` or `showreg` those cluster robust standard errors should then be automatically be shown.

Many other forms of robust standard errors can be computed with the package `sandwich`, but there is no direct support for clustered standard errors. The package `multiwayvcov` allows different versions of clustered standard errors. If you want to show the robust standard errors with stargazer, you have to provide the computed standard errors separately.

For simplicity, the function `showreg` contains the arguments `robust` and `robust.type` or the more general `vcov.li` that allow showing robust standard errors in an output table.

2.4.3 Marginal effects for probit and logit models:

To show marginal effects for probit and logit models use the argument `coef.transform="mfx"` in `showreg`. It based on functions from the package `mfx` (not yet fully tested, though)

2.4.4 Effect sizes: effectplot (in regtools)

Since explanatory variables are usually measured in different units, it is often not easy to compare the sizes of the effects that 'typical' changes of different explanatory variables have on the dependent variable in a regression model. The function `effectplot` in my package `regtools` shall help for such comparison. The key idea is to compare the effects the explanatory variables change from e.g. their 10% quantile to their 90% quantile. For non-linear specifications also the package `effects` is quite helpful, to see how effects change over the range of values of an explanatory variable.

2.4.5 Data preparation: dplyr, tidyr and dplyrExtras

The package `dplyr` and `tidyr` provide a very well thought through framework for many common data manipulation tasks and the functions run fast also for big data sets. Try to use them. Unfortunately, `dplyr` is still very young and I was missing some functionality. Therefore I wrote the package `dplyrExtras` provides that can be installed from Github:

```
library(devtools);
install_github(repo="dplyrExtras", username="skranz")
```

2.4.6 Summarising and aggregating data: dplyr (group_by and summarise)

For summarizing data, the `summarise` function in `dplyr` is very useful. Combine it with the `group_by` function to perform summaries by groups. Also the `summarise_each` function, as well as the `xsummarise_each` function in `dplyrExtras`, can be helpful.

2.4.7 Graphics: ggplot2 and more

A very nice and powerful graphics package is `gpplot2`. Even though the syntax may look a bit complicated at the beginning, I recommend to use it for most of your figures.

The ggplot2 package is nicely complemented by the `ggthemes` package that contains several additional themes for the appearance of your plots. I particularly like `theme_wsj()`, which renders your graphics in a 'Wall Street Journal' theme. There is also a Stata theme: `theme_stata()`, which may make your graphs look more similar to those in the original article.

For simple plots you may also sometimes use the standard R functions like `plot` or `hist`. For interactive plots take a look at the packages `googleVis`. In particular the motion plots in `googleVis` can be quite nice. To use it, set in the chunk header the option `results="asis"` to make the output be displayed as HTML. I have not yet figured out how the promising `ggvis` package can be best integrated with RTutor.

2.5 Formal Aspects

Your interactive problem set should also stick to the following guidelines for its formal aspects. This is especially recommended if you generate it as part of your Bachelor or Master Thesis.

2.5.1 Citations and bibliography

Like in a normal Bachelor thesis, please also you citations. For example, if you explain some econometric procedure you may cite an econometric text book or articles that provide more detailed overviews. Similar to citations in footnotes, you can briefly discuss this literature in an info block. Citation format should be the same as in a normal thesis, e.g.:

```
... see Greene (2011, chapter 7) for more details.
```

At the end of your last exercise you should add a list of all cited literature (a bibliography) using a similar format as you would use in your thesis.

Of course you can also add web links in your problem set. You must not necessarily use the common citation format for web links and may not necessarily add them to bibliography. Decide yourself, whether you think this is sensible or not for a particular link.

2.5.2 Spellchecking

Make sure to run a spell and grammar checker on your _sol.Rmd file.

2.5.2.1 RStudio spell checker

RStudio has a spell checker included. Just press F7 while editing your _sol.Rmd file (or go to menu Edit –> Check Spelling).

2.5.2.2 Using a spell and grammar checker of your Office application

Microsoft Word, for example, has a substantially more powerful spell checker than the one included in RStudio. In particular, it also check your grammar to some extend. You can directly open your .Rmd file in Word and use its spell checker. It probably will mark some of your R code or your markdown annotations as wrong, but you just can manually skip those false alerts.

2.5.3 Code lines not longer than 80 characters

In order for the code chunks being nicely displayed in the HTML version of RTutor, please avoid lines of codes that are longer than 80 characters. You can add manual line breaks in your R code.

2.6 Examples of nice features in existing problem sets

2.6.1 Overall, nicely designed problem set. With good didactical structure of questions

```
- RTutorTopIncomeTaxation
- RTutorSoapOperas
```

2.6.2 Awards

```
- RTutorSoapOperas
```

2.6.3 Interactive footnotes

```
- creditboomsgonebust
```

2.6.4 Nice ggplot2 plots

```
- almost all problem sets
```

2.6.5 Nice lattice barplots

`- creditboomsgonebust 2.1b), 2.2`

2.6.6 Nice googleVis motion plots

`- RTutorTopIncomeTaxation`

2.6.7 Explanation of endogeniety problems and ommited variable problem

`- RTutorSoapOperas`