Customizing and extending Emacs Speaks Statistics

Part of the ESS-Intro series

Frédéric Santos*

March 31, 2021

Contents

*frederic.santos@u-bordeaux.fr

1. Introductory words

1.1. A short tutorial for customizing ESS

This document is a part of the ESS-Intro tutorial series. A full list of tutorials is available online at `https://github.com/ess-intro`.

If you are are totally new to Emacs, you should probably begin with the tutorial "First steps with Emacs" by Dirk Eddelbuettel.

1.1.1. Target audience

This tutorial is for R-users beginning to use Emacs Speaks Statistics (ESS). We only assume that you have mild familiarity with Emacs, and that you know how to customize Emacs by filling your `.emacs` or `init.el` file.

1.1.2. Goals

- Helping newcomers and beginners to have a more friendly and efficient setup
- Helping users migrating from other IDEs to keep behaviours and habits they are used to
- Showcasing some other Emacs packages that play well with ESS and may improve your coding experience

1.1.3. Video tutorial

This document is a written companion for the video tutorial available at:

`https://www.youtube.com/watch?v=Lf8qrLuvYp8`

1.1.4. An opinionated tutorial?

Some parts of this tutorial (e.g., replacing `Flymake` by `Flycheck` for syntax checking, or using `company` instead of other solutions for auto-completion) might be seen as quite opinionated. However, when several options or approaches were available for a given task, the most widespread and efficient (I hope!) was presented.

If you prefer other approaches, or simply want to give a try to some other ones, the ESS manual remains the canonical and exhaustive documentation: `http://ess.r-project.org/Manual/ess.html`

Feedbacks, fixes and contributions are welcome, and can be addressed through the Github repository.

1.2. A word about `use-package`

In this document, several pieces of Emacs Lisp code will be proposed so that you can use them in your init file. Because it will provide a more reproducible setup in the following steps, it is assumed that you use `use-package` for your init file. The Emacs Lisp code can be adapted in a straightforward manner if you do not use it.

As a reminder, this is the minimal code to add in your init file so as to use `use-package`, once it has been installed:

```
;; Make sure that use-package is installed:
(unless (package-installed-p 'use-package)
  (package-refresh-contents)
  (package-install 'use-package))
;; Load use-package:
(eval-when-compile
  (require 'use-package))
```

2. ESS customization

2.1. Visibility of code evaluation

From an R source file (i.e., an ESS[R] buffer), you can send code to the underlying R process with shortcuts like `C-c C-b` (`ess-eval-buffer`), `C-RET` (`ess-eval-region-or-line-and-step`), etc.

In the inferior R process buffer, ESS can display either both the commands sent and their results, or only the results. The choice between the two can be made through the variable `ess-eval-visibly`.

This variable can be `t` (eval visibly and wait for process) or `nil` (eval invisibly); but by default it is set to `'nowait'`, which means that ESS shows input commands in the process buffer, but allows for further editing in the source buffer even while the previous instructions sent are still running. This way, ESS is never freezing while evaluating code.

If you prefer to eval R code invisibly, add in your init file:

```
(setq ess-eval-visibly nil)
```

2.2. Syntactic highlighting in ESS R source buffers

By default, ESS highlights literals, assignments, source functions and reserved words. An example of default syntactic highlighting is as follows:

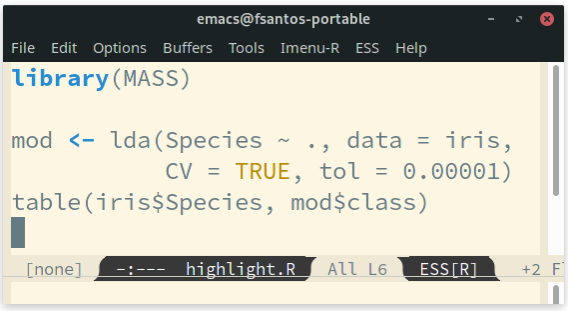

Figure 1: Default patterns for syntactic highlighting.

You can add more patterns to be highlighted, or remove some of them, by customizing the variable `ess-R-font-lock-keywords`.

For instance, if you also want the numbers and the function calls to be highlighted:

```
;; Font lock keywords for syntactic highlighting:
(setq ess-R-font-lock-keywords
      '((ess-R-fl-keyword:keywords . t)
        (ess-R-fl-keyword:constants . t)
        (ess-R-fl-keyword:modifiers . t)
        (ess-R-fl-keyword:fun-defs . t)
        (ess-R-fl-keyword:assign-ops . t)
        (ess-R-fl-keyword:%op% . t)
        (ess-fl-keyword:fun-calls . t)
        (ess-fl-keyword:numbers . t)
        (ess-fl-keyword:operators)
        (ess-fl-keyword:delimiters)
        (ess-fl-keyword:=)
        (ess-R-fl-keyword:F&T . t)))
```

The same code is now highlighted differently, and maybe somewhat more clearly:

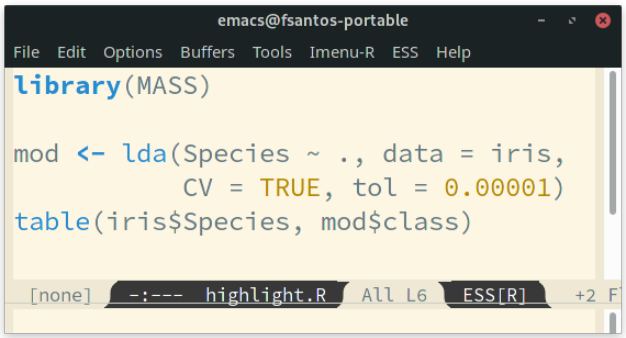

Figure 2: More custom patterns for syntactic highlighting.

2.3. Parenthesis matching

2.3.1. Seeing matching parentheses

Directly taken from the ESS manual (`http://ess.r-project.org/Manual/ess.html#Parens`):

> "Emacs has facilities for highlighting the parenthesis matching the parenthesis at point. This feature is very useful when trying to examine which parentheses match each other. This highlighting also indicates when parentheses are not matching."

To activate parenthesis matching in ESS[R] (source) buffers, add this to your init file:

```
;; Activate global mode for parenthesis matching:
(show-paren-mode)
```

2.3.2. Navigating through matching parentheses

Here are some convenient tricks for navigating through parenthetical groups (this can be useful when dealing with large paren groups, e.g. when developing a shiny UI):

Shortcut	`Elisp function` (Docstring)
`C-M-p`	`backward-list` (Move backward across one balanced paren group)
`C-M-n`	`forward-list` (Move forward across one balanced paren group)
`C-M-SPC`	`mark-sexp` (Set mark at the end of the paren group)
`C-M-k`	`kill-sexp` (Kill from point to end of paren group)

Table 1: Some useful shortcuts for dealing with parenthetical groups.

For instance, when the point is over a closing parenthesis, `C-M-p` brings you to the matching opening parenthesis. Then, `C-M-k` kills to whole paren group.

2.4. Syntax checker

2.4.1. Syntax checking: `Flycheck` vs. `Flymake`

ESS has facilities for on-the-fly syntax checking. Instead of using `Flymake`, which is the default choice, using `Flycheck` appears to be a better and more stable option. The `Flycheck` documentation allows for a comparison between those two packages: `https://www.flycheck.org/en/latest/user/flycheck-versus-flymake.html`

To switch from `Flymake` to `Flycheck`, you can add the following in your init file:

```
;; Remove Flymake support:
(setq ess-use-flymake nil)
;; Replace it (globally) by Flycheck:
(use-package flycheck
  :ensure t
  :init
  (global-flycheck-mode t))
```

2.4.2. On-the-fly syntax checking with `Flycheck`

Using `Flycheck` with ESS first requires you to install the R package `lintr`:

```
## Install stable CRAN version:
install.packages("lintr", dep = TRUE)
## OR
## Install latest Github devel version:
devtools::install_github("jimhester/lintr")
```

(Some users reported that you might also have to create manually a folder `~/.R/lintr_cache` on your computer, if it was not created after the previous step.)

`lintr` is an R package that offers facilities for static code analysis. It integrates with the main IDEs and text editors (Emacs, Rstudio, vim, etc.). In particular, it has native support for ESS + `Flycheck`.

Once both `Flycheck` and `lintr` are installed, your R code is analyzed "on-the-fly" while your are typing. Several checks are performed, including:

- R code style: correct use of `snake_case`, convenient spacing around all operators, etc.
- undeclared variables in function body
- bad use of = for variable assignment
- unmatched parentheses

The following screenshot (Fig. 3) gives some examples of such checks. To display all syntax error in a dedicated buffer, use `M-x flycheck-list-errors` (bound to `C-c ! l` by default).

Figure 3: An example of syntax errors detected by `lintr` and displayed in a dedicated `Flycheck` buffer.

2.5. Some more steps towards an R IDE

2.5.1. Rdired buffer

From the ESS manual:

> "Ess-rdired provides a dired-like buffer for viewing, editing and plotting objects in your current R session. If you are used to using the dired (directory editor) facility in Emacs, this mode gives you similar functionality for R objects."

All the R objects of the current R sessions are thus listed in the Rdired buffer, and it is possible to interact with them easily. For instance, type `p` for plotting an object, `d` for deleting it, etc.

The screenshot in Figure 4 shows the contents of an Rdired buffer for the R session associated to a small piece of code.

Rdired buffers can be triggered manually with `M-x ess-rdired`, which might not be really convenient in practice. With the following piece of Emacs Lisp code, you will be able to use `F9` for both opening and closing the Rdired buffer, so that you can consult and display it only when necessary:

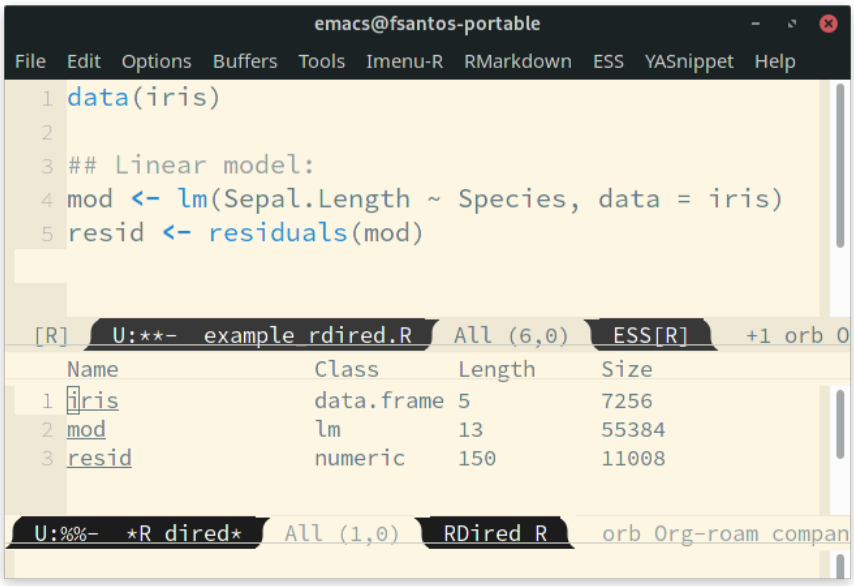

Figure 4: An example of Rdired buffer (bottom window).

```
;; Open Rdired buffer with F9:
(add-hook 'ess-r-mode-hook
          '(lambda ()
             (local-set-key (kbd "<f9>") #'ess-rdired)))
;; Close Rdired buffer with F9 as well:
(add-hook 'ess-rdired-mode-hook
          '(lambda ()
             (local-set-key (kbd "<f9>") #'kill-buffer-and-window)))
```

2.5.2. Window management

Users coming from other R IDEs may be used to a given window (or *pane*) configuration, e.g.:

- R source code window at the top left
- R console (i.e., inferior R process) at the top right
- Rdired environment window at the bottom left
- R help window at the bottom right

This is only an arbitrary example, but quite a reasonable one. You will find in Figure 5 a screenshot of such a window configuration. The corresponding Emacs Lisp code to add in your init file follows.

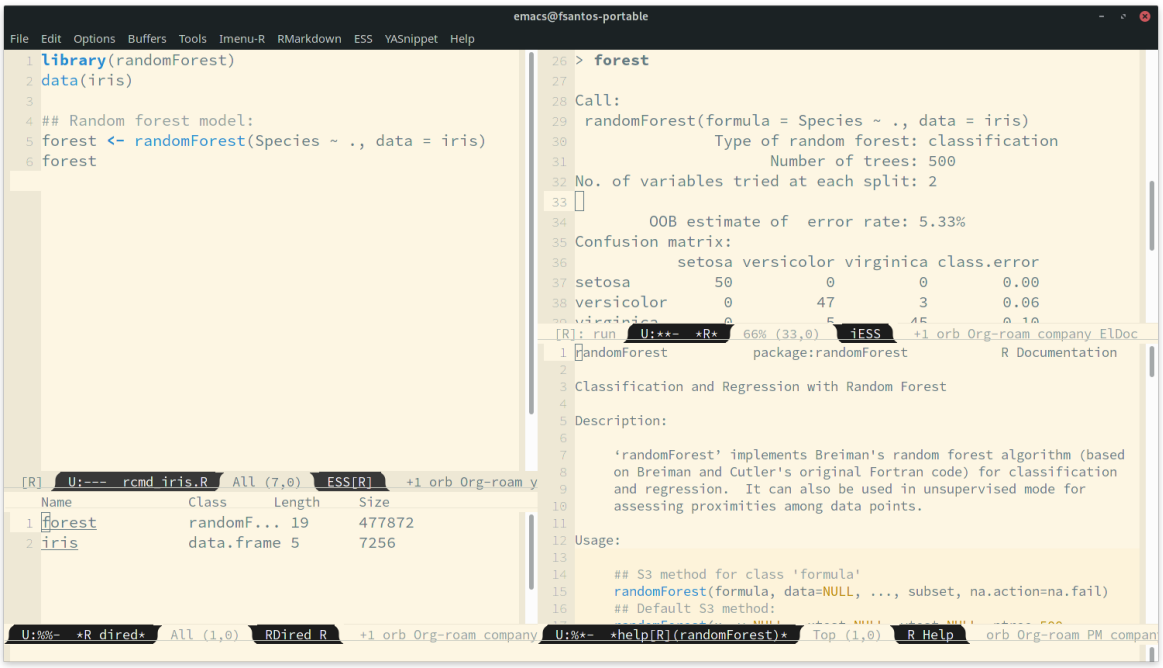

Figure 5: An example of window configuration: ESS as an R IDE.

```
;; An example of window configuration:
(setq display-buffer-alist
      '(("*R Dired"
         (display-buffer-reuse-window display-buffer-at-bottom)
         (window-width . 0.5)
         (window-height . 0.25)
         (reusable-frames . nil))
        ("*R"
         (display-buffer-reuse-window display-buffer-in-side-window)
         (side . right)
         (slot . -1)
         (window-width . 0.5)
         (reusable-frames . nil))
        ("*Help"
         (display-buffer-reuse-window display-buffer-in-side-window)
         (side . right)
         (slot . 1)
         (window-width . 0.5)
         (reusable-frames . nil))))
```

3. Some useful Emacs packages

3.1. Completion with company

As mentioned in the ESS manual, there are several completion frameworks for writing R code with ESS. The Emacs package `company` is an elegant solution, which also supports many other programming languages. Here is a minimal piece of Elisp code to add in your init file to install and load `company`:

```
(use-package company
  :ensure t
  :config
  ;; Turn on company-mode globally:
  (add-hook 'after-init-hook 'global-company-mode)
  ;; Only activate company in R scripts, not in R console:
  (setq ess-use-company 'script-only))
```

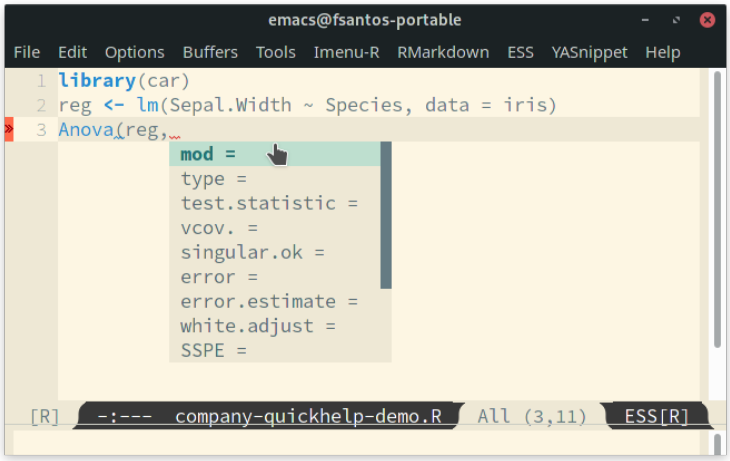

Figure 6: An example of code completion with `company`: various candidates are proposed for the arguments of the function `car::Anova()`.

`company` offers completion candidates in various contexts: function name, argument name within a function call (as in Fig. 6), object name. It may seem preferable to adopt a non-intrusive workflow. For functions or objects names, completion starts automatically after you type a few letters. For arguments names within a function call, it is suggested that you trigger manually the completion only when you need it. This can be done with `M-x company-complete`, or more conveniently, by binding this function to a convenient shortcut. For example, to bind it to `F12`, add the following to your init file:

```
;; Use F12 to trigger manually completion on R function args:
(add-hook 'ess-r-mode-hook
          '(lambda ()
             (local-set-key (kbd "<f12>") #'company-R-args)))
```

Of course, further customization of `company` can be done in your init file. For instance:

```
;; More customization options for company:
(setq company-selection-wrap-around t
      ;; Align annotations to the right tooltip border:
      company-tooltip-align-annotations t
      ;; Idle delay in seconds until completion starts automatically:
      company-idle-delay 0.45
      ;; Completion will start after typing two letters:
      company-minimum-prefix-length 2
      ;; Maximum number of candidates in the tooltip:
      company-tooltip-limit 10)
```

3.2. Documentation popups with company-quickhelp

`company-quickhelp` allows for documentation popups, e.g. to further describe function arguments.

Figure 7: Documentation popups with `company-quickhelp`.

The minimal elisp code to add to your init file is straightforward:

```
(use-package company-quickhelp
  :ensure t
  :config
  ;; Load company-quickhelp globally:
  (company-quickhelp-mode)
  ;; Time before display of documentation popup:
  (setq company-quickhelp-delay 0.3))
```

By default, the documentation popup is shown automatically. You can adjust the time before the popup shows up by customizing the variable `company-quickhelp-delay`.

3.3. Code snippets with yasnippet

3.3.1. Key features

`yasnippet` is an Emacs package allowing for the expansion of whole pieces of code you often use (*snippets*) from one given abbreviation.

- All code snippets are stored as plain-text files in one given directory, so that they are easy to share with other people, and can be easily version controlled.
- As a corollary, it is also easy to retrieve and use large collection of snippets already available online. For instance, Andrea Crotti maintains a great collection available at `https://github.com/AndreaCrotti/yasnippet-snippets`.
- Although we only demonstrate its use within ESS and R here, note that `yasnippet` is not an R-specific solution, and that you can use it for any other programming language.

3.3.2. Setting up `yasnippet`

To set up `yasnippet`, proceed through the following steps:

1. Create a directory `snippets/` at some convenient location, and add a subfolder `ess-r-mode/` in this directory.
2. Add the minimal following code in your init file:

```
(use-package yasnippet
  :ensure t
  :config
  ;; Indicate the directory containing your snippets:
  (setq yas-snippet-dirs '("path/to/your/snippets"))
  ;; Load your snippets on startup:
  (yas-reload-all)
  ;; Turn on yasnippet (minor) mode when editing R files:
  (add-hook 'ess-r-mode-hook #'yas-minor-mode))
```

3. You can now fill your `snippets/ess-r-mode/` directory with your own snippets. For instance, create a file `function` (without any extension) in this directory, with the following contents:

```
#name : function
#key : fun
# --
${1:name} <- function(${2:args}) {
    ${3:body}
}
```

   Each snippet has a unique `name`, and can be triggered by typing a given `key` (followed by `TAB`). As we will see later on, the present snippet allows for the expansion of a template for defining new R functions more easily. The `yasnippet` manual gives more details about the expected syntax to define your own code snippets: `http://joaotavora.github.io/yasnippet/`.

4. Now your `snippets` directory should look like:

```
└── snippets
    └── ess-r-mode
        └── function
```

   Feel free to add or retrieve (a lot!) more snippets, i.e. to add more template files within the `ess-r-mode` sub-directory.

3.3.3. Using `yasnippet` in an ESS[R] buffer

While you are editing an R source file with ESS, each snippet can be triggered by typing its `key` and then pressing `TAB`. You can then navigate through the placeholders of the expanded template by pressing `TAB` again.

For instance, with our previously defined snippet, typing `fun` followed by `TAB` will expand the full `function` template; you will then be able to specify easily a value for each of the three placeholders (the function's `name`, its `args` and `body`).

Note that `yasnippet` has a short video tutorial, available at `https://www.youtube.com/watch?v=ZCGmZK4V7Sg`.