Portfolio Report 3: Projects, version control, and packages

Jake Spiteri

2019

R makes it relatively easy to structure a project, by creating a "project". It also integrates git and github which makes it easy to track changes and publish code to the web.

Projects: organizing code

The best way to approach projects is to maintain a well-defined file structure. RStudio projects allow you to easily divide your work into different directories. Of course, this can be done manually. Generally a good high-level file structure within your project's directory is:

- `R/` — Contains all of the project's R scripts. It is best to not divide these files into subdirectories.
- `README.md` — A markdown document describing the project.
- `data/` — Contains the data used for the project. It may be split into subdirectories `raw/` and `processed/`.
- `doc/` — Contains the documentation for the project. If implementing a method, an academic paper detailing the method may be found here.
- `output/` — The output(s) produced by applying the R scripts in `R/` to the data. The outputs may be divided into subdirectories; all plots may be in a `figures/` folder for example.
- `tests/` — Contains R scripts which test the functions developed in the package. It is best practice to produce test functions which check that functions work as expected. These functions can also be run every time code is changed, so the author is aware when functionality breaks.

Version control with git and GitHub

A version control system that keeps track of the changes made to a set of documents. Each time we track a change, it creates a snapshot of our files which we can revert back to at any point in time. Many of us make copies of our files and store multiple versions of them. Version control systems such as git make this process extremely easy and less prone to errors.

Git

Setup

Once git is installed, it will be useful to set your identity as a global configuration parameter. These parameters can be set for specified git repositories, or globally.

To set your name and email address, use the following commands in the terminal.

```
git config --global user.name "Your Name"
git config --global user.email name@domain.co.uk
```

We can always check the global configuration using the following commands:

```
git config --list --show-origin
```

Otherwise, we can directly view the global parameters using the following

```
git config --global <parameter>
```

Initializing a git repository

We can easily use git to start tracking changes within a directory. Simply navigate to the top-level folder for which you want to track changes in, and type

```
git init
```

The subdirectory will then become a fully-functional git repository. Before git starts to track files you must add them to the staging area.

Adding files to the staging area

A useful command before adding files to the staging area is `git status`. This will show the status of every file within your working directory: what files have been edited, moved, deleted, and which files are being tracked or not. It will also give some advice on how to add files to the staging area, how to discard changes in the current working tree, and how to unstage files.

To add a file to the staging area, use the command

```
git add <filename>
```

Note that you can add entire subdirectories to the staging area using this command. You may encounter a situation in which you have staged a file you do not want to track. In this case you can remove said file from the staging area using

```
git reset HEAD <filename>
```

Similarly to adding files to the staging area, you can unstage individual files or entire subdirectories. To remove everything from the staging area, simply use

```
git reset HEAD .
```

Committing changes

Once a file has been staged, we can commit the file using

```
git commit -m "commit message"
```

It is good practice to commit only files which change a single piece of functionality, rather than commit *all* changes that have been made. It is also useful to add an informative commit message which explains what has changed. The commit saves the changes that have been made to the code, so we can return to a previous version of the code if necessary.

Reverting / Resetting

Sometimes changes have been made locally which have not been commited, which break functionality. In such a case, we may want to revert to the last commit. To do this we can use

```
git revert HEAD
```

We can think of this as undoing changes that have been made. It inverts the last commit and then appends a new commit with this inverted content. It is also important to note the difference between `revert` and `reset`. Firstly, `git revert` changes made in a single commit (this can be *any* commit in the repo's history) whereas `git reset` returns the repo to a previous commit by *removing* all subsequent commits. So `revert` maintains the history of the repo by adding a new commit, and `reset` *deletes* some history by returning to a previous commit and deleting subsequent commits.

Creating a branch

When making substantial changes to an existing repo, it may be better to work on a different 'branch' of the project. We can then work on the branch without affecting the master branch.

To create a new git branch, we use the following

```
git branch <branch-name>
```

To then switch to our new branch we use

```
git checkout <branch-name>
```

This means we can easily switch between the master and existing branches. In order to merge the changes made in the new branch to the master, we switch to the master using `git checkout master` and then merge using `git merge <branch-name>`.

If there are conflicts, git will add comments to the files which will need to be manually addressed before merging.

To delete a branch simply use `git branch -d <branch-name>`.

Check out an old version

You may want to return to an old version of your code. To do this we can get commit hashes using the log `git log`, and then checkout an old snapshot of the code using

```
git checkout <commit hash> -b <branch-name>
```

Packages

Similar to projects, RStudio also makes it easy to create packages. Within a package there are some necessary files such as

- `DESCRIPTION` — Provides details of who created the package, who maintains it, the version of R used etc.
- `LICENSE` — Details the license you have chosen for the package e.g. GPL-2, MIT+, etc.
- `NAMESPACE`
- `R/` — A folder containing the package's R scripts.
- `man` — A folder containing the documentation.
- `tests` — A folder containing scripts that test the package's functionality.

Description file

The first thing you should do after creating a package is edit the DESCRIPTION file. The template provided by RStudio makes this pretty self explanatory. You should add the license, the title, and the author, etc.

Documentation

A good package should be well documented. The package `devtools` and RStudio make documenting code straightforward. You can enable automatic documentation building in RStudio after installing the `devtools` package, by clicking `Build -> Configure Build Toolds... -> Generate documentation with Roxygen`.

You can also directly add documentation for functions in a package by right clicking the function and selecting `Code -> Insert Roxygen skeleton`. You can describe the function's features and parameters in the skeleton above the function, and this will be added to the help document for the function.

Testing

When developing a package it is useful to use a test suite to ensure that functionality doesn't break during development. In order to test the functionality you must write test scripts that compare the output of a function to the expected output. Writing test scripts requires additional effort but they greatly simplify the development process. Testing ensures that the code has fewer bugs as we explicitly test the code's behavior. Testing in RStudio can be done with the `testthat` package. Once this package is installed you should run

```
usethis::testthat("<name>")
```

This will create a new directory called `tests` in which you can add your test scripts. There should already be a test script in the folder called `test-<name>.R`.

Once the test suite is set up the workflow is simple: We modify our code or tests, test the package by running `devtools::test()`, and fix any errors that have occurred.

Chapter 10 of the book R Packages is a great resource for writing test scripts. The book can be found here.

Coverage

When creating a package we want to develop a well-rounded test suite. It may be difficult to ensure that all of our code is covered by our tests — it seems plausible that there are some scenarios which should throw errors which have not been tested by a script. To mitigate this issue we can use tools that tell us which lines have been tested by our test suite. This is called *code coverage*. This is very easy to do in R using the package `covr`. Once installed we can test the coverage of a package by running

```
covr:report()
```

The report provides a lot of useful information: which lines have been covered, which have not been covered, how many times a line is covered, etc. It will also tell us which percentage of the code is covered, and ideally we will strive for 100% coverage. This means that when new code is added to the package we will need to write new test scripts.

Example

An example R package using all of the tools described above can be found here.