Automatic Checks

Thanks to Professor Gerard

Learning Objectives

- Package Checking
- Continuous Integration
- Chapter 19 from R Packages

Package Checking

- `R CMD check` is a command-line tool for automatic package checking that can automatically detect common problems that are hard to check.
- `R CMD check` can be run from R via

```
devtools::check()
```

- The check will return errors, warnings, and notes.
- **Errors**: Serious problems you should fix right away.
- **Warnings**: Likely problems that you should fix eventually.
- **Notes**: Minor problems that may or may not be real problems. I would fix them anyway.
- Workflow for package checking:

    1. Run `devtools::check()`
    2. Fix problem.
    3. Repeat until no more problems.

- I wouldn't run `devtools::check()` every time you make a change. I would do it only a couple times a day. But the earlier you do it, the easier it is to fix all issues.

Common issues

- If you run into an error/warning/note that you do not understand, go to the R packages page to understand it: https://r-pkgs.org/r-cmd-check.html
- I won't cover all of the checks, but I'll list the ones that I most often need to fix.
- In the "DESCRIPTION" file, your title should be in title case ("This is a Title", not "This is a title"), and not end in a period. I would also go through this checklist for the DESCRIPTION file.
- Every argument of a documented function needs to be documented, and there cannot be excess documentation.
  - If you have a typo in your `{roxygen2}` documentation, you will get this warning:

```
> checking Rd \usage sections ... WARNING
  Undocumented arguments in documentation object 'pkg'
    'arg'
  Documented arguments not in \usage in documentation object 'pkg':
    'agr'
```

```
- An undocumented function (where you don't have `{roxygen2}` comments or where you added the
```

- Your package should not be too large.
  - R will tell you which folders have large files.
  - If you are including too much data, that will be an issue.
  - If R tells you that only `libs` is too large then this is usually OK:

```
> checking installed package size ... NOTE
    installed size is 5.2Mb
    sub-directories of 1Mb or more:
      libs 4.4Mb
```

```
- The above often occurs when you are using Rcpp.
```

- Failure to have a package installed: This happens sometimes when you work on multiple computers (or collaborate with others). You can install all dependencies of a package with

```
devtools::install_deps(dependencies = TRUE)
```

- Failure to specify a package:
  - R will tell you if you have not imported a function, but try to use it.
  - The error will look like this:

```
> checking R code for possible problems ... NOTE
  simreg: no visible global function definition for ‘rnorm’
  Undefined global functions or variables:
    rnorm
  Consider adding
    importFrom("stats", "rnorm")
  to your NAMESPACE file.
```

- The R comes with the following packages which are attached at startup by default (from the R FAQ)
  - `{base}`
    - Base R functions (and datasets before R 2.0.0).
  - `{compiler}`
    - R byte code compiler (added in R 2.13.0).
  - `{datasets}`
    - Base R datasets (added in R 2.0.0).
  - `{grDevices}`
    - Graphics devices for base and grid graphics (added in R 2.0.0).
  - `{graphics}`
    - R functions for base graphics.
  - `{grid}`
    - A rewrite of the graphics layout capabilities, plus some support for interaction.
  - `{methods}`
    - Formally defined methods and classes for R objects, plus other programming tools, as described in the Green Book.
  - `{parallel}`
    - Support for parallel computation, including by forking and by sockets, and random-number generation (added in R 2.14.0).
  - `{splines}`
    - Regression spline functions and classes.
  - `{stats}`
    - R statistical functions.
  - `{stats4}`

    - Statistical functions using S4 classes.
  - `{tcltk}`
    - Interface and language bindings to Tcl/Tk GUI elements.
  - `{tools}`
    - Tools for package development and administration.
  - `{utils}`
    - R utility functions.

- Unless a function is from `{base}`, you will need to specify that package using `::`. E.g. `stats::rnorm()` or `utils::read.table()`. Most of the issues I come across is forgetting about the `{stats}` package. You can see what functions are from `{base}` with

```
library(help = "base")
```

Continuous Integration

- You can set up GitHub Actions so that it will run `R CMD check` on multiple virtual machines (Windows, Mac, or Ubuntu) each time you push. This is really great for making sure your package is robust and constantly being checked.
- Automatic checking each time you make a change is called continuous integration.
- In a package, run

```
usethis::use_github_action_check_standard()
```

- Running this will create a new file in a hidden folder via the path ".github/workflows/R-CMD-check.yaml". This YAML file contains instructions for setting up a virtual machine, installing R and your dependencies, and running R CMD check.
- To use it, simply commit your files and push to GitHub, then wait for the checks to run. You can see their progress by clicking on the "Actions" tab on the GitHub page of your package.
- It's not too important to know what that file does, but there are some parts that you may need to edit.
- You may comment out one of the operating systems for the check if you know that the error is artificial. Use `#` for comments in a YAML file. Below, I comment out the Mac.

```
strategy:
  fail-fast: false
  matrix:
```

```
        config:
          - {os: windows-latest, r: 'release'}
          # - {os: macOS-latest, r: 'release'}
          - {os: ubuntu-20.04, r: 'release', rspm: "https://packagemanager.rstudio.com/cra
          - {os: ubuntu-20.04, r: 'devel', rspm: "https://packagemanager.rstudio.com/cran/
```

- Sometimes (but rarely) you need to fix the install code for the dependencies. Onetime `{remotes}` was failing to install the correct Bioconductor packages I needed, so I had to edit it this way:

```
    - name: Install dependencies
      run: |
        remotes::install_deps(dependencies = TRUE)
        remotes::install_cran("rcmdcheck")
        install.packages("BiocManager") # new line
        BiocManager::install("VariantAnnotation") # new line
      shell: Rscript {0}
```

- You can see a variety of other YAML files at https://github.com/r-lib/actions/tree/v1/examples

Exercise

Recall the `simreg()` example from the Testing lecture. Use the edit-check workflow to further develop your package with the following capabilities:

1. In `simreg()`, instead of simulating x from a standard normal, give the user the ability to choose the variance of x, which we will call τ^2.
2. It is probably difficult for the user to specify both σ^2 (the residual variance) and τ^2 (the variance of the predictors). A better option would allow the user to provide more intuitive inputs. One possible input would be the proportion of variance explained (PVE), which we will define as
$$PVE = \frac{\beta_1^2\tau^2}{\beta_1^2\tau^2 + \sigma^2}.$$
This follows from
$$var(y_i) = var(\beta_0 + \beta_1 x_i + \epsilon_i) = \beta_1^2\tau^2 + \sigma^2,$$
and so $\beta_1^2\tau^2$ is how much of the variance in y is explained by the predictors. Allow the user to set the PVE, the residual variance (σ^2), and the regression coefficient (β_1). To do this, you should create a new function called `tau_from_pve()` which will calculate

the proper τ^2 given the PVE, the residual variance, and the regression coefficient. Then you can just use that τ^2 to simulate x.

3. It would probably be better to include many options to choose x. Create a new function called `simx()` that will generate x values under different conditions:

    1. From $N(0, \tau^2)$ after specifying τ
    2. From `sample(x = c(a, b), size = n, replace = TRUE)` for different numeric values of `a` and `b`.