WRITING YOUR OWN R PACKAGES

how, why, and what's in it for me?

Why packages?

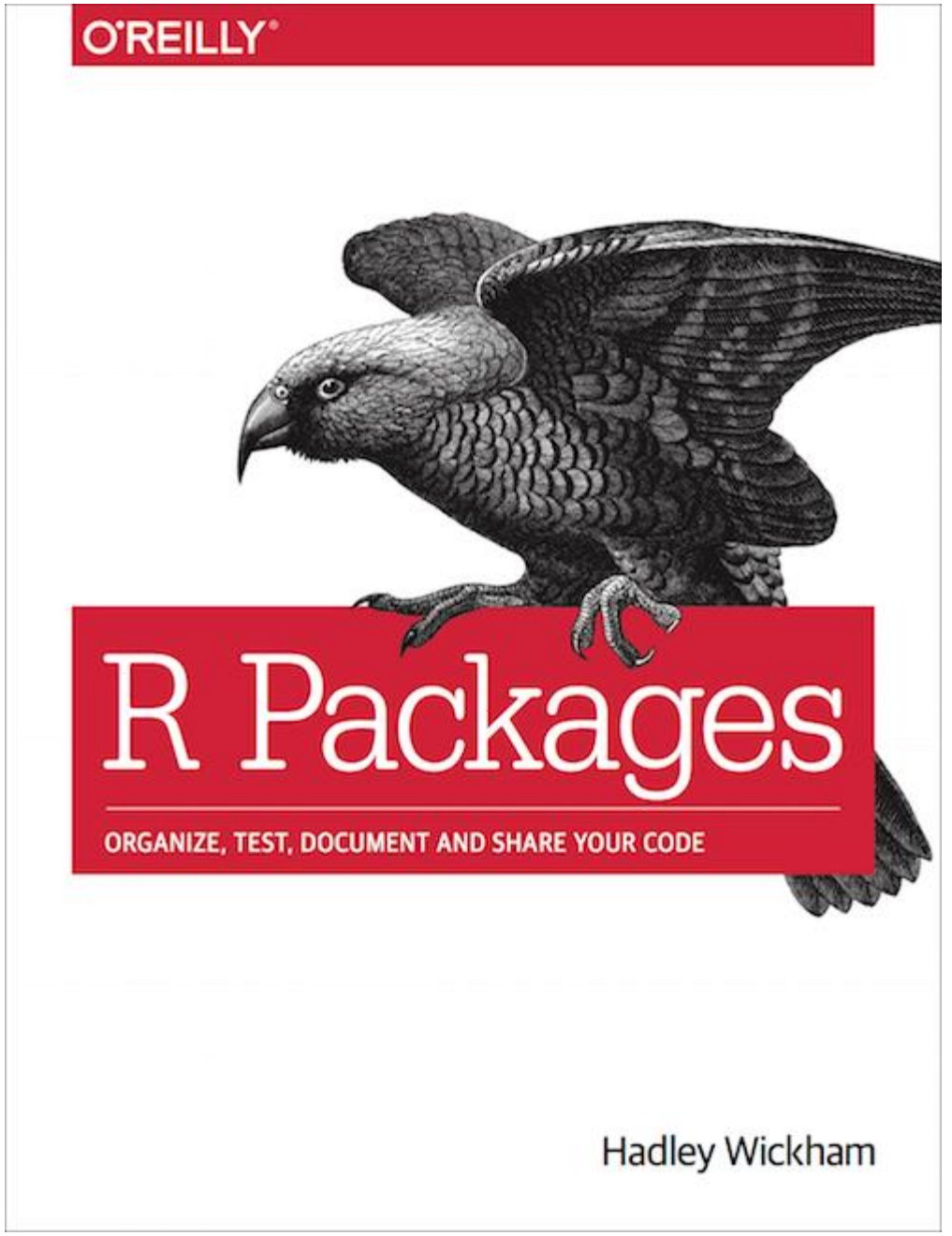
- You have a great idea and others can benefit from it
- You have code that should be shared with and used by coworkers or collaborators
- Can help you organize your code for longer projects
- Can help you document code that you may use intermittently

Soap box

If you do something cool, for the love of all that is holy, share it!

Essential resource

- Book is available online for free at http://r-pkgs.had.co.nz/
- Hadley Wickham updates the online book regularly
- Following the procedures in this book will help you construct a package that can be submitted to CRAN



Packages needed

- Rtools
- devtools
- roxygen2
- testthat
- Most recent versions of R and R Studio

Basic structure

- R – folder contains all of your R Code
- DESCRIPTION - defines metadata for the package
- man folder - roxygen2 produces help pages that explain how to use the package
  - Information added to R code and help pages generated by roxygen2
- NAMESPACE – defines which functions are needed by other packages and what functions it makes available to other packages
  - Automatically generated by roxygen2
- README – Records updates and other helpful information about version changes and other basic information about the package
  - Should be updated every time the package is updated

How to start

- Choose a name
  - Can only contain letters, numbers, and periods
  - Cannot end with a period
  - Make it unique and applicable to what your package does
  - Recommended: use all lowercase letters
- Update R and R Studio
- Two choices
  - Select the following in R Studio from the drop down menus
    - New project > New directory > R package
    - Enter in the name of your package
    - Tell R Studio the directory path of where your package code should be
    - Check boxes for uploading to Github and/or using packrat if desired
  - devtools::create("path/to/package/pkgname")
- Now you have a skeletal version of an R package

\R directory

- CREATE
- All of your R code goes here
- Most of the work for the package happens in these files
- Thoughtfully organize your functions into files
  - Simple package: one function per file
  - Complex package: several functions per file
- If you have a hard time finding your functions
  - Use better names for files and functions
  - Consider changing file configuration

DESCRIPTION

- **Edit** the DESCRIPTION file that magically appears in your package
- Contains metadata for the package
- Include:
  - Title
  - Author(s)
  - Description
  - Required packages
  - Recommended packages
  - License - most packages use GPL-3

```
Package: mypackage
Title: What The Package Does (one line, title case required)
Version: 0.1
Authors@R: person("First", "Last", email = "first.last@example.com",
                  role = c("aut", "cre"))
Description: What the package does (one paragraph)
Depends: R (>= 3.1.0)
License: What license is it under?
LazyData: true
Description: The description of a package is usually long,
    spanning multiple lines. The second and subsequent lines
    should be indented, usually with four spaces.
Imports:
    dplyr,
    ggvis
Suggests:
    dplyr,
    ggvis
```

\man folder

- DO NOT EDIT THE FILES IN THIS FOLDER!
- Contains .Rd files that create the help files
- roxygen2 automatically creates .Rd files from information included in the .R files
- Create .Rd files using the following commands
  - devtools::load_all()
  - devtools::document()

\man folder

.R file

```r
#' Cross-Validated Accuracy for Supervised Learning Model
#'
#' eztune.cv is a function that will return the cross-validated
#' accuracy for a model generated by the function eztune.
#' The function eztune can tune a model using resubstitution or
#' cross-validation. If the resubstitution is used to tune the model, the
#' accuracy obtained from the function is inflated. The function
#' eztune.cv will return a cross-validated accuracy for such a model.
#' @param x Matrix or data frame containing the dependent variables used
#' to create the model.
#' @param y Numeric vector of 0s and 1s for the response used to create
#' the model.
#' @param model Object generated with the function eztune.
#' @param fold Number of folds to use for n-fold cross validation.
#' @keywords adaboost, svm, gbm, tuning, cross-validation
#' @return Function returns a numeric value that represents the
#' cross-validated accuracy of the model.
#'
#' @examples
#' library(mlbench)
#' data(Glass)
#'
#' glass <- Glass[as.numeric(as.character(Glass$Type)) < 3, ]
#' glass <- glass[sample(1:nrow(glass), 80), ]
#' y <- ifelse(glass$Type == 1, 0, 1)
#' x <- glass[, 1:9]
#'
#' glass_svm <- eztune(x, y, type = "binary", method = "svm")
#' eztune.cv(x, y, glass_svm)
#'
#' @export
#'
eztune.cv <- function(x, y, model, fold = 10) {


  param <- switch(class(model$best.model)[1],
                  ada = c(model$iter, model$nu),
                  gbm = c(model$interaction.depth, model$n.trees,
                          model$shrinkage),
                  svm.formula = c(model$epsilon, model$cost))
```

.Rd file

```r
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/eztune.cv.R
\name{eztune.cv}
\alias{eztune.cv}
\title{Cross-Validated Accuracy for Supervised Learning Model}
\usage{
eztune.cv(x, y, model, fold = 10)
}
\arguments{
\item{x}{Matrix or data frame containing the dependent variables used
to create the model.}

\item{y}{Numeric vector of 0s and 1s for the response used to create
the model.}

\item{model}{Object generated with the function eztune.}

\item{fold}{Number of folds to use for n-fold cross validation.}
}
\value{
Function returns a numeric value that represents the
cross-validated accuracy of the model.
}
\description{
eztune.cv is a function that will return the cross-validated
accuracy for a model generated by the function eztune.
The function eztune can tune a model using resubstitution or
cross-validation. If the resubstitution is used to tune the model, the
accuracy obtained from the function is inflated. The function
eztune.cv will return a cross-validated accuracy for such a model.
}
\examples{
library(mlbench)
data(Glass)

glass <- Glass[as.numeric(as.character(Glass$Type)) < 3, ]
glass <- glass[sample(1:nrow(glass), 80), ]
y <- ifelse(glass$Type == 1, 0, 1)
x <- glass[, 1:9]
```

Help page

eztune.cv {EZtune} R Documentation

Cross-Validated Accuracy for Supervised Learning Model

Description

eztune.cv is a function that will return the cross-validated accuracy for a model generated by the function eztune. The function eztune can tune a model using resubstitution or cross-validation. If the resubstitution is used to tune the model, the accuracy obtained from the function is inflated. The function eztune.cv will return a cross-validated accuracy for such a model.

Usage

```
eztune.cv(x, y, model, fold = 10)
```

Arguments

x	Matrix or data frame containing the dependent variables used to create the model.
y	Numeric vector of 0s and 1s for the response used to create the model.
model	Object generated with the function eztune.
fold	Number of folds to use for n-fold cross validation.

Value

Function returns a numeric value that represents the cross-validated accuracy of the model.

Examples

```
library(mlbench)
data(Glass)
```

NAMESPACE

- DO NOT EDIT!
- Created automatically by roxygen2 using information .R files
- Contains imports and exports
- NAMESPACE files can get very complex and they are difficult to understand.
  - Roxygen2 will give you what you need
  - Eventually, you may want to edit your file

README.md

- You need to create and update this
- Should be updated every time you update the package
- Includes
  - Information about past versions
  - Changes made to current version
  - Notes that are helpful for you
  - Notes that are helpful for others
- Everything will work fine without a README file, but it's bad form to not have one and maintain it

Other features

- Data (data/)
  - You can include data files as part of your package
- Other compiled code (src/)
  - You may use C or C++ to execute tasks in your package
  - Can use Rcpp
  - Can import and export C or C++ code
- Vignette (vignettes/)
  - If you write a really nice vignette you should publish it

Other features

- Installed files (inst/)
  - Author
  - Citation
  - Additional external data
  - Java, python, perl, ruby, etc. code
- Other components
  - Package demos (demo/)
  - Executable scripts (exec/)
  - Translation messages (po/)
  - Auxillary files needed during configuration (tools/)

Coding your package

- How do your write the R files?
  - Create .R files and save them to R/
  - Each exported functions (the ones package users will use) in their own files
    - This is not necessary, but it is much cleaner
  - Keep hidden functions in separate files from exported functions
  - Use roxygen2 comments in the exported functions to produce documentation
  - Remember to include examples
- Test code often using devtools::load_all() and then using your package as intended
- Run devtools::document() to create help files
  - Test help files using ?package_function

Best practices

- ALWAYS use the package::function() notation when coding
- NEVER have your package install other packages
  - This should be taken care of in the DESCRIPTION file
- ALWAYS create and maintain a README.md file, even if you are the only one using the package
- FREQUENTLY reassess your file structure and reorganize your functions if the current configuration no longer makes sense
- Write good examples
- Write helpful error and warning messages
- Provide thorough documentation even if you are the only one using the package
  - Comment all code
  - Make good help files

More best practices

- Use google style guide for R coding
- Write all packages as if you are going to publish them
- Github is your friend
  - Sharing your code with other is a great way to find and fix bugs
  - Share development versions
- TIP: NAMESPACE can be tricky if you are having trouble with it
  - Delete the NAMESPACE file and run devtools::document()
  - Make sure R and R Studio have permission to write to your computer

Getting ready to make code public

- Test your own code carefully before publishing it on Github or CRAN
- Run R CMD check on the code
  - devtools::check()
- Test package on at least two operating systems
  - Windows
  - Linux or Unix
  - Mac OS
- TIP: sometimes the devtools functions fail to run the first time. Try running them again before assuming there is something wrong

Github

- You can upload a package to github even if you don't initially tell R to do it
- Great way to share code so others can use it and comment on it
- devtools can be used to install packages directly from github

```
library(devtools)
install_github("author/package")
```

Example:

```
install_github("jillbo1000/EZtune")
```

Submitting to CRAN

- Submitting to CRAN is a bit daunting, but worth it if you did something really useful and you want people to use it
- CRAN is very picky
  - Using Hadley Wickham's book will make the process as painless as possible
- Run devtools::check() in two operating systems before submission
  - Get rid of all errors, warnings, and notes
- First submission is automated
- Plan on several submissions
  - Three guys have to check your code in their spare time without pay
  - Make sure you get your package as perfect as possible before each submission
  - They will get cranky if you don't or if you submit revisions too often