



CHEATSHEETS

Last update: 2022-08-02



A compilation of R cheatsheets taken from
<https://www.rstudio.com/resources/cheatsheets/>

Table of contents

advancedr.....	1	oscr.....	48
bayesplot.....	6	overviewr.....	51
bcea.....	7	package-development.....	52
caret.....	8	packagefinder.....	54
cartography.....	9	parallel-computation.....	55
collapse.....	10	plumber.....	56
data-import.....	11	purrr.....	58
data-transformation.....	13	quanteda.....	60
data-visualization.....	15	quincunx.....	62
datatable.....	17	randomizr.....	64
declaredesign.....	19	regex.....	65
distr6.....	20	reticulate.....	66
estimatr.....	22	rmarkdown.....	68
eurostat.....	23	rphylopic.....	70
factors.....	24	rstudio-ide.....	71
gganimate.....	25	samplingstrata.....	73
golem.....	27	sas-r.....	75
gwasrapid.....	28	sf.....	77
h2o.....	29	shiny.....	79
how-big-is-your-graph.....	31	sparklyr.....	81
imputets.....	33	strings.....	83
jfa.....	34	survminer.....	85
keras.....	35	syntax.....	86
labelled.....	37	teachr.....	88
leaflet.....	39	tidy়.....	89
lubridate.....	40	time-series.....	91
machine-learning-modelli..	42	tsbox.....	92
mlr.....	43	vegan.....	93
nardl.....	45	vtree.....	94
nimble.....	46	xplain.....	95

Advanced R Cheat Sheet

Created by: Arianne Colton and Sean Chen

Environment Basics

Environment – **Data structure** (with two components below) that powers lexical scoping

Create environment: `env1<-new.env()`

- Named list** (“Bag of names”) – each name points to an object stored elsewhere in memory.

If an object has no names pointing to it, it gets automatically deleted by the garbage collector.

- Access with: `ls('env1')`

- Parent environment** – used to implement lexical scoping. If a name is not found in an environment, then R will look in its parent (and so on).

- Access with: `parent.env('env1')`

Four special environments

- Empty environment** – ultimate ancestor of all environments

- Parent: none
- Access with: `emptyenv()`

- Base environment** - environment of the base package

- Parent: empty environment
- Access with: `baseenv()`

- Global environment** – the interactive workspace that you normally work in

- Parent: environment of last attached package
- Access with: `globalenv()`

- Current environment** – environment that R is currently working in (may be any of the above and others)

- Parent: empty environment
- Access with: `environment()`

Environments

Search Path

Search path – mechanism to look up objects, particularly functions.

- Access with: `search()` – lists all parents of the global environment (see Figure 1)
- Access any environment on the search path: `as.environment('package:base')`

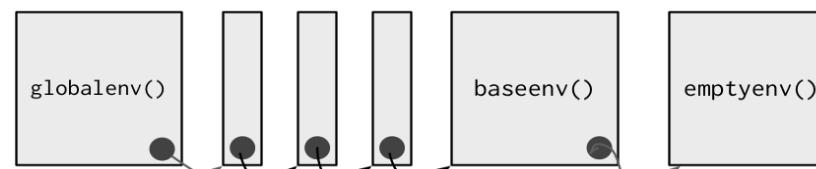


Figure 1 – The Search Path

- Mechanism : always start the search from global environment, then inside the latest attached package environment.
 - New package loading with `library()`/`require()` : new package is attached right after global environment. (See Figure 2)
 - Name conflict in two different package : functions with the same name, latest package function will get called.

`search()` :

```

'.GlobalEnv' ... 'Autoloads' 'package:base'
library(reshape2); search()
'.GlobalEnv' 'package:reshape2' ... 'Autoloads' 'package:base'
  
```

NOTE: Autoloads : special environment used for saving memory by only loading package objects (like big datasets) when needed

Figure 2 – Package Attachment

Binding Names to Values

Assignment – act of binding (or rebinding) a name to a value in an environment.

- `<-` (Regular assignment arrow) – always creates a variable in the current environment
- `<<-` (Deep assignment arrow) - modifies an existing variable found by walking up the parent environments

Warning: If `<<-` doesn't find an existing variable, it will create one in the global environment.

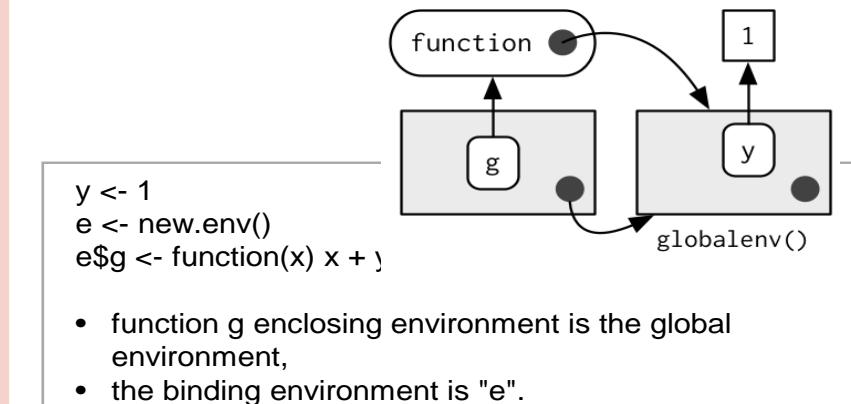
Function Environments

- Enclosing environment** - an environment where the function is created. It determines how function finds value.
 - Enclosing environment never changes, even if the function is moved to a different environment.
 - Access with: `environment('func1')`

- Binding environment** - all environments that the function has a binding to. It determines how we find the function.

- Access with: `pryr::where('func1')`

Example (for enclosing and binding environment):



- Execution environment** - new created environments to host a function call execution.

- Two parents :
 - Enclosing environment of the function
 - Calling environment of the function
- Execution environment is thrown away once the function has completed.

- Calling environment** - environments where the function was called.

- Access with: `parent.frame('func1')`
- Dynamic scoping :
 - About : look up variables in the calling environment rather than in the enclosing environment
 - Usage : most useful for developing functions that aid interactive data analysis

Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Note: R has no 0-dimensional or scalar types. Individual numbers or strings, are actually vectors of length one, NOT scalars.

Human readable description of any R data structure :

```
str(variable)
```

Every **Object** has a mode and a class

1. **Mode**: represents how an object is stored in memory

- 'type' of the object from R's point of view
- Access with: **typeof()**

2. **Class**: represents the object's abstract type

- 'type' of the object from R's object-oriented programming point of view
- Access with: **class()**

	typeof()	class()
strings or vector of strings	character	character
numbers or vector of numbers	numeric	numeric
list	list	list
data.frame	list	data.frame

Factors

1. Factors are built on top of integer vectors using two attributes :

```
class(x) -> 'factor'
```

```
levels(x) # defines the set of allowed values
```

2. Useful when you know the possible values a variable may take, even if you don't see all values in a given dataset.

Warning on Factor Usage:

1. Factors look and often behave like character vectors, they are actually integers. Be careful when treating them like strings.
2. Most data loading functions automatically convert character vectors to factors. (Use argument `stringAsFactors = FALSE` to suppress this behavior)

Object Oriented (OO) Field Guide

Object Oriented Systems

R has three object oriented systems :

1. **S3** is a very casual system. It has no formal definition of classes. It implements generic function OO.

- **Generic-function OO** - a special type of function called a generic function decides which method to call.

Example:	<code>drawRect(canvas, 'blue')</code>
Language:	R

- **Message-passing OO** - messages (methods) are sent to objects and the object determines which function to call.

Example:	<code>canvas.drawRect('blue')</code>
Language:	Java, C++, and C#

2. **S4** works similarly to S3, but is more formal. Two major differences to S3 :

- **Formal class definitions** - describe the representation and inheritance for each class, and has special helper functions for defining generics and methods.
- **Multiple dispatch** - generic functions can pick methods based on the class of any number of arguments, not just one.

3. **Reference classes** are very different from S3 and S4:

- **Implements message-passing OO** - methods belong to classes, not functions.
- **Notation** - \$ is used to separate objects and methods, so method calls look like `canvas$drawRect('blue')`.

S3

1. About S3 :

- R's first and simplest OO system
- Only OO system used in the base and stats package
- Methods belong to functions, not to objects or classes.

2. Notation :

- **generic.class()**

<code>mean.Date()</code>	Date method for the generic - <code>mean()</code>
--------------------------	---

3. Useful 'Generic' Operations

- Get all methods that belong to the 'mean' generic:
 - **Methods('mean')**
- List all generics that have a method for the 'Date' class :
 - **methods(class = 'Date')**

4. **S3 objects** are usually built on top of lists, or atomic vectors with attributes.

- Factor and data frame are S3 class
- Useful operations:

Check if object is an S3 object	<code>is.object(x) & !isS4(x) or pryr::otype()</code>
Check if object inherits from a specific class	<code>inherits(x, 'classname')</code>
Determine class of any object	<code>class(x)</code>

Base Type (C Structure)

R base types - the internal C-level types that underlie the above OO systems.

- **Includes** : atomic vectors, list, functions, environments, etc.
- **Useful operation** : Determine if an object is a base type (Not S3, S4 or RC) `is.object(x)` returns FALSE

• **Internal representation** : C structure (or struct) that includes :

- Contents of the object
- Memory Management Information
- Type
- Access with: **typeof()**

Functions

Function Basics

Functions – objects in their own right

All R functions have three parts:

body()	code inside the function
formals()	list of arguments which controls how you can call the function
environment()	“map” of the location of the function’s variables (see “Enclosing Environment”)

Every operation is a function call

- +, for, if, [, \$, { ...
- x + y is the same as `+`(x, y)

Note: the backtick (`), lets you refer to functions or variables that have otherwise reserved or illegal names.

Lexical Scoping

What is Lexical Scoping?

- Looks up value of a symbol. (see “Enclosing Environment”)
- **findGlobals()** - lists all the external dependencies of a function

```
f <- function() x + 1
codetools::findGlobals(f)
> '+'
environment(f) <- emptyenv()
f()
# error in f(): could not find function "+"
```

- R relies on lexical scoping to find everything, even the + operator.

Function Arguments

Arguments – passed by reference and copied on modify

1. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.
2. Check if an argument was supplied : **missing()**

```
i <- function(a, b) {
  missing(a) -> # return true or false
}
```

3. Lazy evaluation – since x is not used **stop("This is an error!")** never get evaluated.

```
f <- function(x) {
  10
}
f(stop('This is an error!')) -> 10
```

4. Force evaluation

```
f <- function(x) {
  force(x)
  10
}
```

5. Default arguments evaluation

```
f <- function(x = ls()) {
  a <- 1
  x
}
```

f() -> 'a' 'x'

ls() evaluated inside f

f(ls())

ls() evaluated in global environment

Return Values

- **Last expression evaluated or explicit return().**
Only use explicit return() when returning early.
- **Return ONLY single object.**
Workaround is to return a list containing any number of objects.
- **Invisible return object value** - not printed out by default when you call the function.

```
f1 <- function() invisible(1)
```

Primitive Functions

What are Primitive Functions?

1. Call C code directly with **.Primitive()** and contain no R code

```
print(sum) :
> function (... , na.rm = FALSE) .Primitive('sum')
```

2. **formals()**, **body()**, and **environment()** are all NULL
3. Only found in base package
4. More efficient since they operate at a low level

Influx Functions

What are Influx Functions?

1. Function name comes in between its arguments, like + or -
2. All user-created infix functions must start and end with %.

```
`%+%` <- function(a, b) paste0(a, b)
'new' %+% 'string'
```

3. Useful way of providing a default value in case the output of another function is NULL:

```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

Replacement Functions

What are Replacement Functions?

1. Act like they modify their arguments in place, and have the special name xxx <-
2. Actually create a modified copy. Can use **pryr::address()** to find the memory address of the underlying object

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
```

Subsetting

Subsetting returns a copy of the original data, NOT copy-on modified

Simplifying vs. Preserving Subsetting

1. Simplifying subsetting

- Returns the **simplest** possible data structure that can represent the output

2. Preserving subsetting

- Keeps the structure of the output the **same** as the input.
- When you use `drop = FALSE`, it's preserving

	Simplifying*	Preserving
Vector	<code>x[[1]]</code>	<code>x[1]</code>
List	<code>x[[1]]</code>	<code>x[1]</code>
Factor	<code>x[1:4, drop = T]</code>	<code>x[1:4]</code>
Array	<code>x[1,] or x[, 1]</code>	<code>x[1, , drop = F] or x[, 1, drop = F]</code>
Data frame	<code>x[, 1] or x[[1]]</code>	<code>x[, 1, drop = F] or x[1]</code>

Simplifying behavior varies slightly between different data types:

1. Atomic Vector

- `x[[1]]` is the same as `x[1]`

2. List

- `[]` always returns a list
- Use `[[]]` to get list contents, this returns a single value piece out of a list

3. Factor

- Drops any unused levels but it remains a factor class

4. Matrix or Array

- If any of the dimensions has length 1, that dimension is dropped

5. Data Frame

- If output is a single column, it returns a vector instead of a data frame

Data Frame Subsetting

Data Frame – possesses the **characteristics of both lists and matrices**. If you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices

1. Subset with a single vector : Behave like lists

```
df1[c('col1', 'col2')]
```

2. Subset with two vectors : Behave like matrices

```
df1[, c('col1', 'col2')]
```

The results are the same in the above examples, however, results are different if subsetting with only one column. (see below)

1. Behave like matrices

```
str(df1[, 'col1']) -> int [1:3]
```

- Result: the result is a vector

2. Behave like lists

```
str(df1['col1']) -> 'data.frame'
```

- Result: the result remains a data frame of 1 column

\$ Subsetting Operator

1. About Subsetting Operator

- Useful shorthand for `[[` combined with character subsetting

```
x$y is equivalent to x[['y', exact = FALSE]]
```

2. Difference vs. `[[`

- \$ does partial matching, `[[` does not

```
x <- list(abc = 1)
x$a -> 1      # since "exact = FALSE"
x[['a']] ->    # would be an error
```

3. Common mistake with \$

- Using it when you have the name of a column stored in a variable

```
var <- 'cyl'
x$var
# doesn't work, translated to x[['var']]
# Instead use x[[var]]
```

Examples

1. Lookup tables (character subsetting)

```
x <- c('m', 'f', 'u', 'f', 'f', 'm', 'm')
lookup <- c(m = 'Male', f = 'Female', u = NA)
lookup[x]
> m f u f f m m
> 'Male' 'Female' NA 'Female' 'Female' 'Male' 'Male'
unname(lookup[x])
> 'Male' 'Female' NA 'Female' 'Female' 'Male' 'Male'
```

2. Matching and merging by hand (integer subsetting)

Lookup table which has multiple columns of information:

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame(
  grade = 3:1,
  desc = c('Excellent', 'Good', 'Poor'),
  fail = c(F, F, T)
)
```

First Method

```
id <- match(grades, info$grade)
info[id, ]
```

Second Method

```
rownames(info) <- info$grade
info[as.character(grades), ]
```

3. Expanding aggregated counts (integer subsetting)

- Problem:** a data frame where identical rows have been collapsed into one and a count column has been added
- Solution:** `rep()` and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index: `rep(x, y)` rep replicates the values in x, y times.

```
df1$countCol is c(3, 5, 1)
rep(1:nrow(df1), df1$countCol)
> 1 1 1 2 2 2 2 2 3
```

4. Removing columns from data frames (character subsetting)

There are two ways to remove columns from a data frame:

Set individual columns to NULL	<code>df1\$col3 <- NULL</code>
Subset to return only columns you want	<code>df1[c('col1', 'col2')]</code>

5. Selecting rows based on a condition (logical subsetting)

- This is the most commonly used technique for extracting rows out of a data frame.

```
df1[df1$col1 == 5 & df1$col2 == 4, ]
```

Subsetting continued

Boolean Algebra vs. Sets (Logical and Integer Subsetting)

1. **Using integer subsetting** is more effective when:

- You want to find the first (or last) TRUE.
- You have very few TRUEs and very many FALSEs; a set representation may be faster and require less storage.

2. **which()** - conversion from boolean representation to integer representation

```
which(c(T, F, T F)) -> 1 3
```

- Integer representation length : is always <= boolean representation length
- Common mistakes :
 - I. Use **x[which(y)]** instead of **x[y]**
 - II. **x[-which(y)]** is not equivalent to **x[!y]**

Recommendation:

Avoid switching from logical to integer subsetting unless you want, for example, the first or last TRUE value

Subsetting with Assignment

1. All subsetting operators can be combined with assignment to modify selected values of the input vector.

```
df1$col1[df1$col1 < 8] <- 0
```

2. Subsetting with nothing in conjunction with assignment :

- Why : Preserve original object class and structure

```
df1[] <- lapply(df1, as.integer)
```

Debugging, Condition Handling and Defensive Programming

Debugging Methods

1. traceback() or RStudio's error inspector

- Lists the sequence of calls that lead to the error

2. browser() or RStudio's breakpoints tool

- Opens an interactive debug session at an arbitrary location in the code

3. options(error = browser) or RStudio's "Rerun with Debug" tool

- Opens an interactive debug session where the error occurred
- Error Options:

options(error = recover)

- Difference vs. 'browser': can enter environment of any of the calls in the stack

options(error = dump_and_quit)

- Equivalent to 'recover' for non-interactive mode
- Creates **last.dump.rda** in the current working directory

In batch R process :

```
dump_and_quit <- function() {
  # Save debugging info to file
  last.dump.rda
  dump.frames(to.file = TRUE)
  # Quit R with error status
  q(status = 1)
}
options(error = dump_and_quit)
```

In a later interactive session :

```
load("last.dump.rda")
debugger()
```

Condition Handling of Expected Errors

1. Communicating potential problems to users:

I. stop()

- Action : raise fatal error and force all execution to terminate
- Example usage : when there is no way for a function to continue

II. warning()

- Action : generate warnings to display potential problems
- Example usage : when some of elements of a vectorized input are invalid

III. message()

- Action : generate messages to give informative output
- Example usage : when you would like to print the steps of a program execution

2. Handling conditions programmatically:

I. try()

- Action : gives you the ability to continue execution even when an error occurs

II. tryCatch()

- Action : lets you specify handler functions that control what happens when a condition is signaled

```
result = tryCatch(code,
  error = function(c) "error",
  warning = function(c) "warning",
  message = function(c) "message"
)
```

Use **conditionMessage(c)** or **c\$message** to extract the message associated with the original error.

Defensive Programming

Basic principle : "fail fast", to raise an error as soon as something goes wrong

1. **stopifnot()** or use 'assertthat' package - check inputs are correct
2. **Avoid subset(), transform() and with()** - these are non-standard evaluation, when they fail, often fail with uninformative error messages.
3. **Avoid [and sapply()** - functions that can return different types of output.
 - Recommendation : Whenever subsetting a data frame in a function, you should always use **drop = FALSE**

Bayesplot :: CHEAT SHEET



```
library("bayesplot")
library("rstanarm")
options(mc.cores = parallel::detectCores())
library("ggplot2")
library("dplyr")
```

Model Parameters

To showcase bayesplot, we'll fit linear regression using `rstanarm::stan_glm` and use this model throughout.

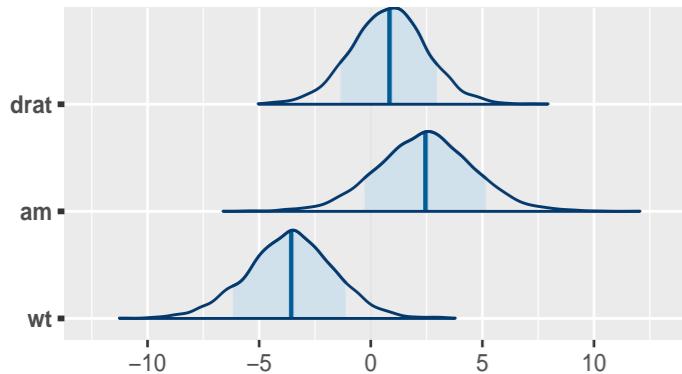
```
model <- stan_glm(mpg ~ ., data=mtcars, chains=4)
posterior <- as.matrix(model)
```

Chances are good you're most interested in the posterior distributions for select parameters.

```
plot_title <- ggtitle("Posterior distributions",
                      "medians and 80% intervals")
mcmc_areas(posterior,
            pars = c("drat", "am", "wt"),
            prob = 0.8) + plot_title
```

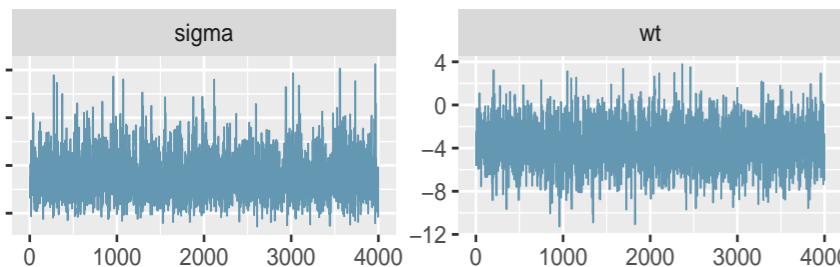
Posterior distributions

medians and 80% intervals



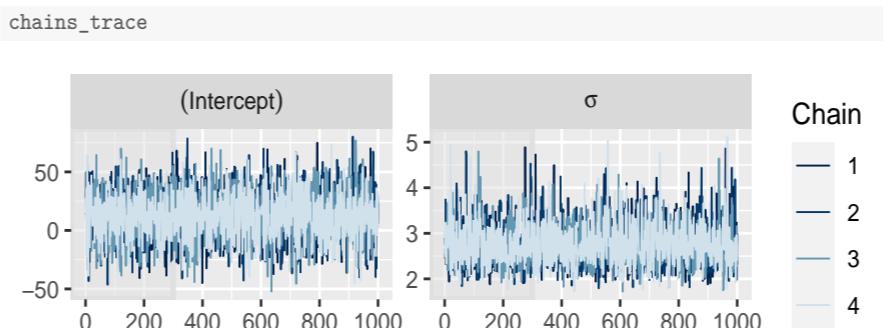
Diagnosing convergence with traceplots is simple.

```
mcmc_trace(posterior, pars=c("sigma", "wt"))
```



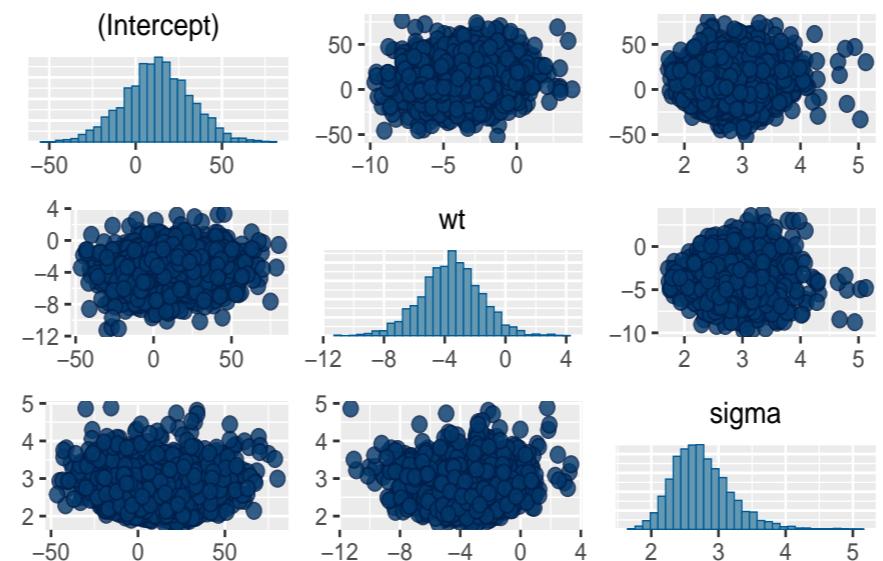
Using `as.array`, you can extract each of the four chain's posterior draws, different from above. This allows you to see each chain's traceplot for selected parameters.

```
posterior_chains <- as.array(model)
fargs <- list(ncol = 2, labeller = label_parsed)
pars <- c("(Intercept)", "sigma")
chains_trace <- mcmc_trace(posterior_chains, pars = pars,
                           n_warmup = 300, facet_args = fargs)
```



The pairs plot is helpful in determining if you have any highly correlated parameters.

```
posterior_chains %>%
  mcmc_pairs(pars = c("(Intercept)", "wt", "sigma"))
```

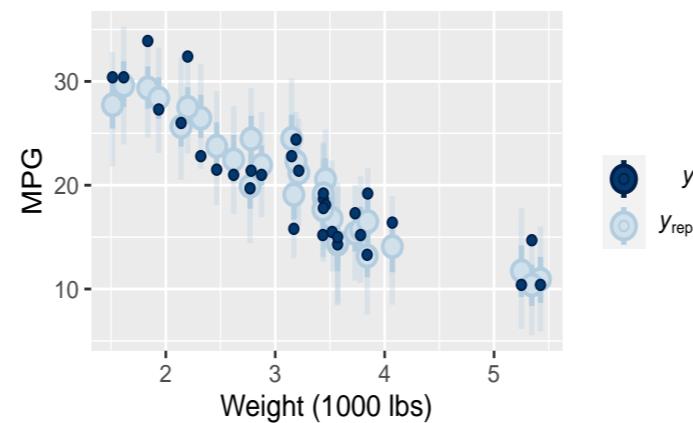


Posterior Predictive Checks

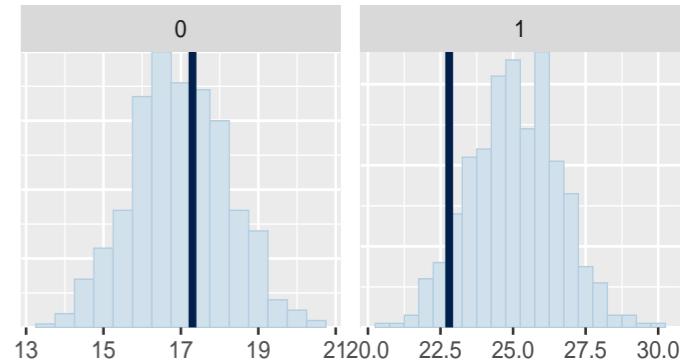
Check how well the model covers your data with draws from the posterior predictive density.

```
ppd <- posterior_predict(model, draws=500)
ppd %>%
  ppc_intervals(y = mtcars$mpg, yrep = ., x = mtcars$wt, prob = 0.5) +
  labs(x = "Weight (1000 lbs)", y = "MPG",
       title = "50% posterior predictive intervals of MPG by weight")
```

50% posterior predictive intervals of MPG by



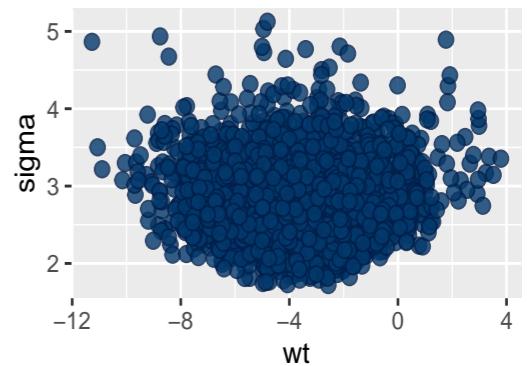
```
ppd %>%
  ppc_stat_grouped(y = mtcars$mpg, group = mtcars$am,
                     stat = "median", binwidth=0.5)
```



Diagnostics

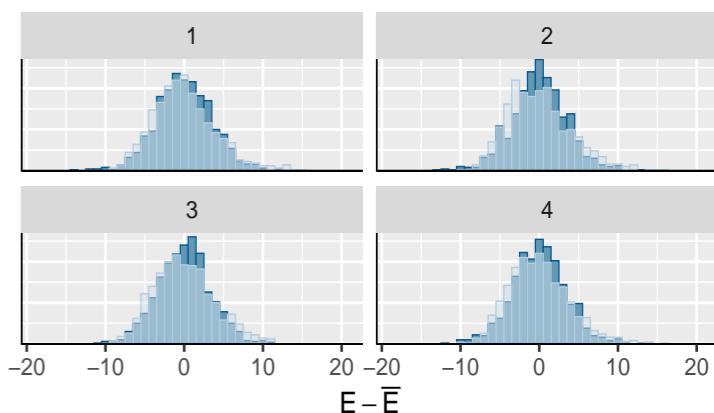
Bayesplot makes it easy to check diagnostics specific to the NUTS sampling `rstanarm` uses by default.

```
mcmc_scatter(posterior, pars = c("wt", "sigma"),
              np = nuts_params(model$stanfit))
```



```
np <- nuts_params(model$stanfit)
mcmc_nuts_energy(np, binwidth=1) +
  ggtitle("NUTS Energy Diagnostic")
```

NUTS Energy Diagnostic



Introduction

Bayesian Cost-Effectiveness Analysis in R

Given a random sample of suitable variables of costs (*cost*) and clinical benefits (*eff*) for two or more interventions produces a health economic evaluation. Inputs may be the results of a Bayesian model (possibly based on MCMC) in the form of simulations from the posterior distributions. For *s* sample points compares one of the *m* interventions (*reference*) to the others (*.comparison*).

```
bcea(eff, cost, ref, .comparison, interventions)
```

INPUT ARRAY PAIR

	cost	eff
Interv1		
Interv2		
Interv3		
<i>s</i>		
<i>m</i>		

bcea()

CONSTITUENT FUNCTIONS

- compute_U()** : Expected utility for each WTP & intervention
- compute_Ustar()** : Maximum ‘known-distribution’ utility for each WTP
- compute_vi()** : Value of information for each WTP
- compute_ol()** : Opportunity Loss for each WTP
- compute_ICER()** : Incremental cost-effectiveness ratio
- compute_IB()** : Incremental benefit for each WTP
- compute_CEAC()** : Cost-effectiveness acceptability for each WTP
- compute_EIB()** : Expected incremental benefit for each WTP
- compute_kstar()** : WTP break-even value

bcea() calculates numerous cost-effectiveness analysis statistics. These can be called directly, using the constituent functions, but would require some pre-processing which is already handled by **bcea()**.

Value assignment

There are 3 equivalent ways to assign values to analysis parameters.

1. *In Constructor*: When first creating a bcea object.

```
he <- bcea(eff, cost, ref, .comparison, ...)
```

2. *Using Setters*: Change directly using replacement functions.

```
setComparison(he) <- comparison
setKmax(he) <- Kmax
setReference(he) <- ref
```

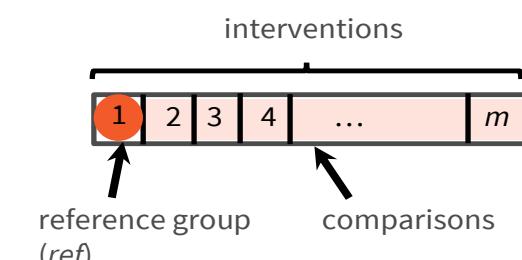
3. *In plotting call*: At the point of making a plot.

```
eib.plot(he, comparison, ...)
ceac.plot(he, comparison, ...)
ceplane.plot(he, comparison, ...)
```

SELECTING ANALYSIS INTERVENTIONS

Default

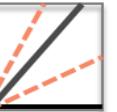
The first columns in (*eff*, *cost*) are the default reference intervention. All other interventions are the comparison interventions unless defined otherwise. E.g. for *m* interventions



Plot

Standard cost-effectiveness analysis output plots. Base R, ggplot2 and plotly versions of plots are available and can be called directly but require extra default parameters.

Expected incremental benefit


eib.plot(he, comparison = NULL, pos = c(1, 0), size = NULL, plot.cri = NULL, graph = c("base", "ggplot2", "plotly"), ...)
calls: • **eib_plot_base()**
• **eib_plot_ggplot()**
• **eib_plot_plotly()**

Expected value of information


evi.plot(he, graph = c("base", "ggplot2", "plotly"), ...)

Cost-effectiveness planes with contours


contour[2](he, comparison = 1, scale = 0.5, nlevels = 4, levels = NULL, pos = c(1, 0), xlim = NULL, ylim = NULL, graph = c("base", "ggplot2"), ...)

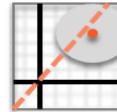
Compare optimal scenario to mixed case


plot.mixedAn(x, y.limits = NULL, pos = c(0, 1), graph = c("base", "ggplot2"), ...)

Cost-effectiveness acceptability curve


ceac_plot(he, comparison = NULL, pos = c(1, 0), graph = c("base", "ggplot2", "plotly"), ...)
calls: • **ceac_plot_base()**
• **ceac_plot_ggplot()**
• **ceac_plot_plotly()**

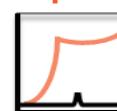
Cost-effectiveness plane


ceplane_plot(he, comparison = NULL, pos = c(1, 0), graph = c("base", "ggplot2", "plotly"), ...)
calls: • **ceplane_plot_base()**
• **ceplane_plot_ggplot()**
• **ceplane_plot_plotly()**

Grid of CE plane, EIB, EVI & CEAC


plot.bcea(x, comparison = NULL, pos = c(1, 0), graph = c("base", "ggplot2", "plotly"), ...)

Expected value of perfect partial information


plot.evppi(x, pos = c(0, 0.8), graph = c("base", "ggplot2"), col = NULL, ...)

Summarise data

Summary statistics and formatted tables can be used to interrogate a **bcea()** object.

summary.bcea(he, ...)

Prints a table with summary results of the health economic evaluation

summary.mixedAn(he, ...)

Prints summary table for results of mixed analysis

sim.table(he, ...)

Summary table of simulations from the cost-effectiveness analysis

make.report(he, ...)

Constructs the automated report from the output of the BCEA

Cheat Sheet

Specifying the Model

Possible syntaxes for specifying the variables in the model:

```
train(y ~ x1 + x2, data = dat, ...)
train(x = predictor_df, y = outcome_vector, ...)
train(recipe_object, data = dat, ...)
```

- `rfe`, `sbf`, `gafs`, and `safs` only have the `x/y` interface.
- The `train` formula method will **always** create dummy variables.
- The `x/y` interface to `train` will not create dummy variables (but the underlying model function might).

Remember to:

- Have column names in your data.
- Use factors for a classification outcome (not 0/1 or integers).
- Have valid R names for class levels (not "0"/"1")
- Set the random number seed prior to calling `train` repeatedly to get the same resamples across calls.
- Use the `train` option `na.action = na.pass` if you will be imputing missing data. Also, use this option when predicting new data containing missing values.

To pass options to the underlying model function, you can pass them to `train` via the ellipses:

```
train(y ~ ., data = dat, method = "rf",
      # options to `randomForest`:
      importance = TRUE)
```

Parallel Processing

The `foreach` package is used to run models in parallel. The `train` code does not change but a “`do`” package must be called first.

```
# on Mac OS or Linux      # on Windows
library(doMC)              library(doParallel)
registerDoMC(cores=4)       cl <- makeCluster(2)
                           registerDoParallel(cl)
```

The function `parallel::detectCores` can help too.

the predictors with the `preProc` option.

```
train(), preProc = c("method1", "method2"), ...)
```

Methods include:

- `"center"`, `"scale"`, and `"range"` to normalize predictors.
- `"BoxCox"`, `"YeoJohnson"`, or `"expoTrans"` to transform predictors.
- `"knnImpute"`, `"bagImpute"`, or `"medianImpute"` to impute.
- `"corr"`, `"nzv"`, `"zv"`, and `"conditionalX"` to filter.
- `"pca"`, `"ica"`, or `"spatialSign"` to transform groups.

`train` determines the order of operations; the order that the methods are declared does not matter.

The `recipes` package has a more extensive list of preprocessing operations.

Adding Options

Many `train` options can be specified using the `trainControl` function:

```
train(y ~ ., data = dat, method = "cubist",
      trControl = trainControl(<options>))
```

Resampling Options

`trainControl` is used to choose a resampling method:

```
trainControl(method = <method>, <options>)
```

Methods and options are:

- `"cv"` for K-fold cross-validation (`number` sets the # folds).
- `"repeatedcv"` for repeated cross-validation (`repeats` for # repeats).
- `"boot"` for bootstrap (`number` sets the iterations).
- `"LGOCV"` for leave-group-out (`number` and `p` are options).
- `"L00"` for leave-one-out cross-validation.
- `"oob"` for out-of-bag resampling (only for some models).
- `"timeslice"` for time-series data (options are `initialWindow`, `horizon`, `fixedWindow`, and `skip`).

function is used again.

```
trainControl(summaryFunction = <R function>,
             classProbs = <logical>)
```

Custom R functions can be used but `caret` includes several: `defaultSummary` (for accuracy, RMSE, etc), `twoClassSummary` (for ROC curves), and `prSummary` (for information retrieval). For the last two functions, the option `classProbs` must be set to `TRUE`.

Grid Search

To let `train` determine the values of the tuning parameter(s), the `tuneLength` option controls how many values **per tuning parameter** to evaluate.

Alternatively, specific values of the tuning parameters can be declared using the `tuneGrid` argument:

```
grid <- expand.grid(alpha = c(0.1, 0.5, 0.9),
                      lambda = c(0.001, 0.01))
```

```
train(x = x, y = y, method = "glmnet",
      preProc = c("center", "scale"),
      tuneGrid = grid)
```

Random Search

For tuning, `train` can also generate random tuning parameter combinations over a wide range. `tuneLength` controls the total number of combinations to evaluate. To use random search:

```
trainControl(search = "random")
```

Subsampling

With a large class imbalance, `train` can subsample the data to balance the classes them prior to model fitting.

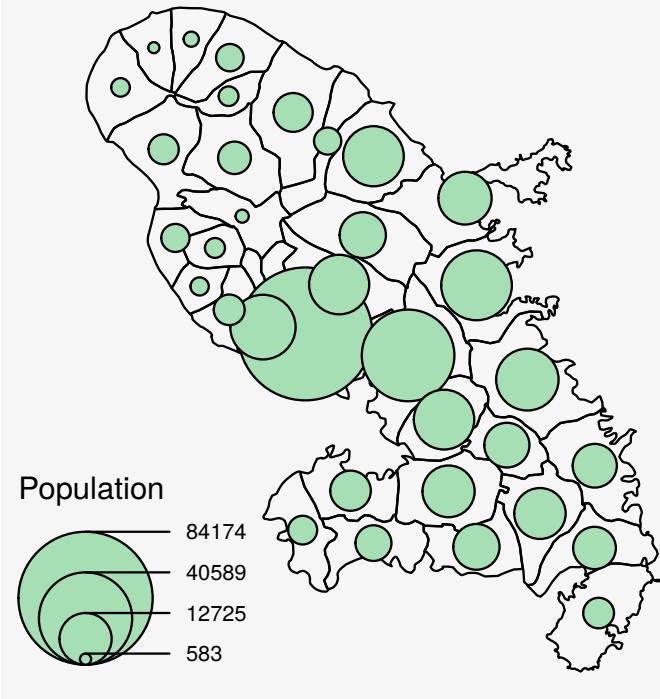
```
trainControl(sampling = "down")
```

Other values are `"up"`, `"smote"`, or `"rose"`. The latter two may require additional package installs.

Thematic maps with cartography :: CHEAT SHEET

Use cartography with spatial objects from sf or sp packages to create thematic maps.

```
library(cartography)
library(sf)
mtq <- st_read("martinique.shp")
plot(st_geometry(mtq))
propSymbolsLayer(x = mtq, var = "P13_POP",
  legend.title.txt = "Population",
  col = "#a7dfb4")
```



Classification

Available methods are: quantile, equal, q6, fisher-jenks, mean-sd, sd, geometric progression...

```
bks1 <- getBreaks(v = var, nclass = 6,
  method = "quantile")
bks2 <- getBreaks(v = var, nclass = 6,
  method = "fisher-jenks")
pal <- carto.pal("green.pal", 3, "wine.pal", 3)
hist(var, breaks = bks1, col = pal)
```



```
hist(var, breaks = bks2, col = pal)
```



Symbology

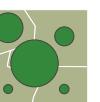
In most functions the x argument should be an sf object. sp objects are handled through spdf and df arguments.



Choropleth
choroLayer(x = mtq, var = "myvar",
method = "quantile", nclass = 8)



Typology
typoLayer(x = mtq, var = "myvar")



Proportional Symbols
propSymbolsLayer(x = mtq, var = "myvar",
inches = 0.1, symbols = "circle")



Colorized Proportional Symbols (relative data)
propSymbolsChoroLayer(x = mtq, var = "myvar",
var2 = "myvar2")



Colorized Proportional Symbols (qualitative data)
propSymbolsTypoLayer(x = mtq, var = "myvar",
var2 = "myvar2")



Double Proportional Symbols
propTrianglesLayer(x = mtq, var1 = "myvar",
var2 = "myvar2")



OpenStreetMap Basemap (see rosm package)
tiles <- getTiles(x = mtq, type = "osm")
tilesLayer(tiles)



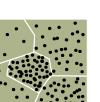
Isopleth (see SpatialPosition package)
smoothLayer(x = mtq, var = "myvar",
typefct = "exponential", span = 500,
beta = 2)



Discontinuities
discLayer(x = mtq.borders, df = mtq,
var = "myvar", threshold = 0.5)



Flows
propLinkLayer(x = mtq_link, df = mtq_df,
var = "fij")



Dot Density
dotDensityLayer(x = mtq, var = "myvar")

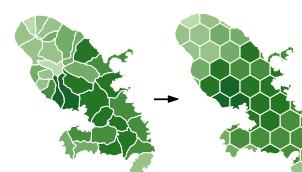


Labels
labelLayer(x = mtq, txt = "myvar",
halo = TRUE, overlap = FALSE)

Transformations

Polygons to Grid

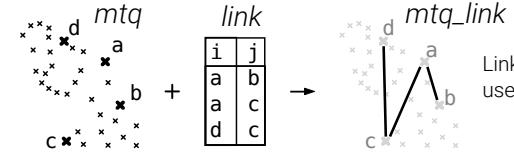
```
mtq_grid <- getGridLayer(x = mtq, cellsize = 3.6e+07,
  type = "hexagonal", var = "myvar")
```



Grids layers can be used by
choroLayer() or propSymbolsLayer().

Points to Links

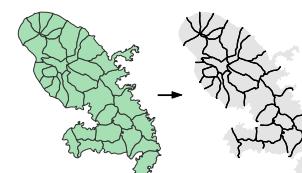
```
mtq_link <- getLinkLayer(x = mtq, df = link)
```



Links layers can be
used by *LinkLayer().

Polygons to Borders

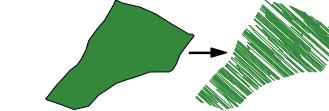
```
mtq_border <- getBorders(x = mtq)
```



Borders layers can be used by
discLayer() function

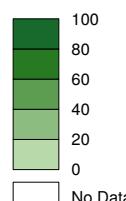
Polygons to Pencil Lines

```
mtq_pen <- getPencilLayer(x = mtq)
```



Legends

legendChoro()



```
legendChoro(pos = "topleft",
  title.txt = "legendChoro()",  
breaks = c(0, 20, 40, 60, 80, 100),  
col = carto.pal("green.pal", 5),  
nodata = TRUE, nodata.txt = "No Data")
```

legendTypo()

```
legendTypo(title.txt = "legendTypo()",  
col = c("peru", "skyblue", "gray77"),  
categ = c("type 1", "type 2", "type 3"),  
nodata = FALSE)
```

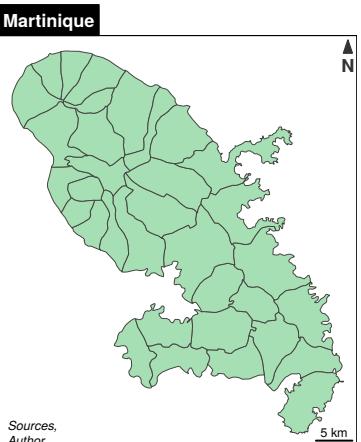
legendCirclesSymbols()

```
legendCirclesSymbols(var = c(10, 100),  
title.txt = "legendCirclesSymbols()",  
col = "#a7dfb4ff", inches = 0.3)
```

See also legendSquaresSymbols(), legendBarsSymbols(),
legendGradLines(), legendPropLines() and legendPropTriangles().

Map Layout

North Arrow:
north(pos = "topright")



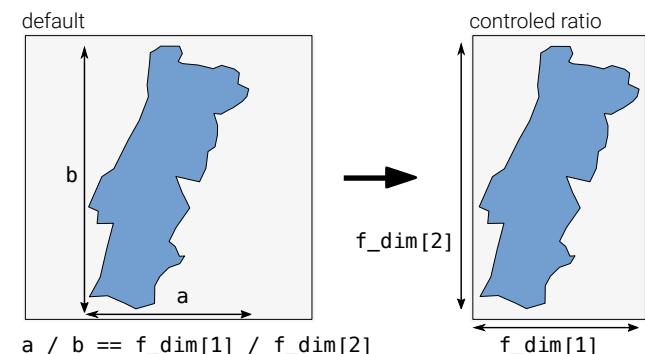
Scale Bar:
barscale(size = 5)

Full Layout:
layoutLayer(
 title = "Martinique",
 tabtitle = TRUE,
 frame = TRUE,
 author = "Author",
 sources = "Sources",
 north = TRUE,
 scale = 5)

Figure Dimensions

Get figure dimensions based on the dimension ratio of a spatial object, figure margins and output resolution.

```
f_dim <- getFigDim(x = sf_obj, width = 500,
  mar = c(0, 0, 0, 0))
png("fig.png", width = 500, height = f_dim[2])
par(mar = c(0, 0, 0, 0))
plot(sf_obj, col = "#729fcf")
dev.off()
```



Color Palettes

carto.pal(pal1 = "blue.pal", n1 = 5,
pal2 = sand.pal, n2 = 3)



display.carto.all(n = 8)





Advanced and Fast Data Transformation with collapse :: CHEAT SHEET

Basics

collapse is a powerful (C/C++ based) package supporting advanced (grouped, weighted, time series, panel data and recursive) operations in R.

It also offers a fast, class-agnostic approach to data manipulation - handling matrix and data frame based objects in a uniform, non-destructive way.

It is well integrated with *dplyr* ((grouped) tibbles), *data.table*, *sf* and *plm* classes for panel data, and can be programmed using pipes (%>%), |>, standard or non-standard evaluation.

Fast Statistical Functions

Fast functions to perform column-wise grouped and weighted computations on matrix-like objects:

```
fmean, fmedian, fmode, fsum, fprod, fsd,
fvar, fmin, fmax, fnth, ffirrst, flast,
fnobs, fndistinct
```

Syntax:

```
FUN(x, g = NULL, [w = NULL], TRA = NULL,
     [na.rm = TRUE], use.g.names = TRUE,
     [drop = TRUE])
```

x – vector, matrix, or (grouped) data frame

g – [optional]: (list of) vectors / factors or **GRP()** object

w – [optional]: vector of weights

TRA – [optional]: operation to transform data with computed statistics (can also be done in post, see section below)

Examples:

```
fmean(data[3:5], data$grp1, data$weights)
data %>% fgroup_by(grp1) %>% fmean(weights)
```

Using dplyr grouped tibble & centering on the median:

```
data %>% dplyr::group_by(grp1) %>%
  fmedian(weights, TRA = "-")
```

Transform by (Grouped) Replacing or Sweeping out Statistics

```
TRA(x, STATS, FUN = '-', g = NULL)
```

STATS – statistics matching columns of x
(e.g. aggregated matrix or data frame)

FUN – string indicating transformation to perform:

- ‘replace_fill’ – overwrite values with statistic
- ‘replace’ – same but keep missing values in data,
- ‘-’ – center, ‘-+’ – center on overall average statistic,
- ‘/’ – scale / divide, ‘%’ – percentages, ‘+’ – add,
- ‘*’ – multiply, ‘%%’ – modulus, ‘-%’ – flatten

Examples:

```
TRA(mat, fmedian(mat, g), "-", g)
fmedian(mat, g, TRA = "-") – same thing
```

Advanced Transformations

Fast functions to perform common and specialized data transformations (for panel data econometrics)

Scaling, (Quasi-)Centering and Averaging

```
fscale(x, g = NULL, w = NULL, na.rm = TRUE,
       mean = 0, sd = 1, ...)
fwithin(x, g = NULL, w = NULL, na.rm = TRUE,
        mean = 0, theta = 1, ...)
fbetween(x, g = NULL, w = NULL, na.rm = TRUE,
          fill = FALSE, ...)
```

High-Dimensional Centering and Averaging

```
fhdwithin(x, fl, w = NULL, na.rm = TRUE,
           variable.wise = FALSE, ...)
fhdbetween(x, fl, w = NULL, na.rm = TRUE,
            fill = FALSE, variable.wise = FALSE, )
```

Operators (function shortcuts with extra features):

```
STD(), W(), B(), HDW(), HDB()
```

Linear Models

```
fIm(y, X, w = NULL, add.icpt = FALSE, method =
c('lm', 'solve', 'qr', 'arma', 'chol', 'eigen'), )
– fast (barebones) linear model fitting with 6 different solvers
```

```
fFtest(y, exc, X = NULL, w = NULL, ...)
– fast F-test of exclusion restrictions for lm's (with HD FE)
```

Time Series and Panel Series

Fast functions to perform time-based computations on (irregular) time series and (unbalanced) panel data

Lags / Leads, Differences and Growth Rates

```
flag(x, n = 1, g = NULL, t = NULL, fill = NA, )
fdiff(x, n = 1, diff = 1, g = NULL, t = NULL,
      fill = NA, log = FALSE, rho = 1, ...)
fgrowth(x, n = 1, diff = 1, g = NULL, t = NULL,
         fill = NA, logdiff = FALSE,
         scale = 100, power = 1, ...)
```

Operators: L(), F(), D(), Dlog(), G()

Cumulative Sums: fcumsum(x, g, o, na.rm, fill,)

Panel-ACF/PACF/CCF | Panel-Data → Array

```
psacf(), pspacf(), psccf() | psmat()
```

Other Computations

Apply functions to rows or columns (by groups)

```
dapply(x, FUN, ..., MARGIN = 2) – column/row apply
BY(x, g, FUN, ...) – split-apply-combine computing
```

Advanced Data Aggregation

Fast multi-data-type, multi-function, weighted, parallelized and fully customized data aggregation

```
collap(data, by, FUN = fmean, catFUN = fmode,
       cols = NULL, w = NULL, wFUN = fsum,
       custom = NULL, ...)
```

Where:

- by – one- or two-sided formula ([vars] ~ groups) or data (like g)
- FUN – (list of) functions applied to numeric columns in data
- catFUN – (list of) functions applied to categorical columns
- cols – [optional]: columns to aggregate (if by is one-sided)
- w – [optional]: one-sided formula or vector giving weights
- wFUN – (list of) functions to aggregate weights passed to w
- custom – [alternatively]: list mapping functions to columns e.g. list(fmean = 1:3, fsum = 4:5, ...)

Examples:

```
collap(data, var1 + var2 ~ grp1 + grp2)
collap(data, ~ grp1, fmedian, w = ~ weights)
collap supports grouped data frames and NS eval:
data %>% gby(grp1) %>% collap(w = weights)
```

Grouping and Ordering

Optimized functions for grouping, ordering, unique values, and for creating and interacting factors

GRP(data, ~ grp1 + grp2) – create a grouping object (class ‘GRP’) from grp1 and grp2 – can be passed to g argument – useful for programming and C/C++ development

fgroup_by(data, grp1, grp2) – attach ‘GRP’ object to data – a flexible grouped data frame that preserves the attributes of data and supports fast computations

fgroup_vars(), fungroup() – get group vars & ungroup

qF(), qG() – quick conversion to factor and vector grouping object (a factor-light class ‘qG’)

groupid() – fast run-length-type group id (class ‘qG’)

seqid() – group-id from integer-sequences (class ‘qG’)

radixorder[v]() – fast Radix-based ordering

finteraction() – fast factor interactions

fdroplevels() – fast removal of unused factor levels

funique() – fast unique values / rows (by cols)

Quick Conversions

qDF(), qDT(), qTBL() – convert vectors, arrays, data.frames or lists to data.frame, data.table or tibble

qMC() – to matrix, m[r/c]t1() – matrix rows/cols to list

as_numeric_factor(), as_character_factor()

– convert factors or all factors in a list / data.frame

Fast Data Manipulation

```
fselect[<-]() – select/replace cols
fsubset() – subset data (rows and cols)
colorder[v]() – reorder cols ('v' FUN's aid programming)
roworder[v]() – sort (reorder) rows
[f/set]transform[v][<-]() – transform cols (by reference)
fcompute[v]() – compute new cols dropping existing ones
[f/set]rename() – rename (any object with 'names' attr.)
get_vars[<-]() – select/replace cols (standard evaluation)
num_vars[<-](), cat_vars[<-](), char_vars[<-]()
fact_vars[<-](), logi_vars[<-]()
date_vars[<-]() – select/replace cols by data type
add_vars[<-]() – add (column - bind) cols
```

List-Processing

Functions to process (nested) lists (of data objects)

```
ldepth() – level of nesting of list
is_unlistable() – is list composed of atomic objects
has_elem() – search if list contains certain elements
get_elem() – pull out elements from list / subset list
atomic_elem[<-](), list_elem[<-]() – get list with atomic / sub-list elements, examining only first level of list
reg_elem(), irreg_elem() – get full list tree leading to atomic ('regular') or non-atomic ('irregular') elements
rsplit() – efficient (recursive) splitting
rapply2d() – recursive apply to lists of data objects
unlist2d() – recursive row-binding to data.frame
```

Summary Statistics

```
qsu() – fast (grouped, weighted, panel-decomposed) summary statistics for cross-sectional and panel data
descr() – detailed statistical description of data.frame
varying() – check variation within groups (panel-id's)
pwcor(), pwcov(), pwobs() – pairwise correlations, covariance and obs. (with P-value and pretty printing)
```

Recode and Replace Values

```
recode_num(), recode_char() – recode numeric / character values (+ regex recoding) in matrix-like objects
replace_NA(), replace_Inf(), replace_outliers() – replace special values
pad() – add observations / rows.
```

Utility Functions

```
.c, vlabels[<-], namlab, na_rm, na.omit,
allNA, missing_cases, ckmatch, add_stub,
rm_stub, fnrow, seq_row, %!in%, unattrib etc...
```

Read Tabular Data with readr

```
read_*(file, col_names = TRUE, col_types = NULL, col_select = NULL, id = NULL, locale, n_max = Inf,
      skip = 0, na = c("", "NA"), guess_max = min(1000, n_max), show_col_types = TRUE) See ?read_delim
```

A B C 1 2 3 4 5 NA	→	A B C 1 2 3 4 5 NA
--------------------------	---	--------------------------------------

read_delim("file.txt", delim = "|") Read files with any delimiter. If no delimiter is specified, it will automatically guess.
To make file.txt, run: write_file("A|B|C\n1|2|3\n4|5|NA", file = "file.txt")

A,B,C 1,2,3 4,5,NA	→	A B C 1 2 3 4 5 NA
--------------------------	---	--------------------------------------

read_csv("file.csv") Read a comma delimited file with period decimal marks.
write_file("A,B,C\n1,2,3\n4,5,NA", file = "file.csv")

A;B;C 1,5;2;3 4,5;NA	→	A B C 1.5 2 3 4.5 5 NA
----------------------------	---	--

read_csv2("file2.csv") Read semicolon delimited files with comma decimal marks.
write_file("A;B;C\n1,5;2;3\n4,5;5;NA", file = "file2.csv")

A B C 1 2 3 4 5 NA	→	A B C 1 2 3 4 5 NA
--------------------------	---	--------------------------------------

read_tsv("file.tsv") Read a tab delimited file. Also **read_table()**.
read_fwf("file.tsv", fwf_widths(c(2, 2, NA))) Read a fixed width file.
write_file("A\tB\tC\n1\t2\t3\n4\t5\tNA", file = "file.tsv")

USEFUL READ ARGUMENTS

A B C 1 2 3 4 5 NA	No header read_csv("file.csv", col_names = FALSE)
--------------------------------------	---

x y z A B C 1 2 3 4 5 NA	Provide header read_csv("file.csv", col_names = c("x", "y", "z"))
---	---

→	Read multiple files into a single table read_csv(c("f1.csv", "f2.csv", "f3.csv"), id = "origin_file")
---	---

A B C 1 2 3 4 5 NA	→	A,B,C 1,2,3 4,5,NA
--------------------------------------	---	--------------------------

Save Data with readr

```
write_*(x, file, na = "NA", append, col_names, quote, escape, eol, num_threads, progress)
```

write_delim(x, file, delim = " ") Write files with any delimiter.
write_csv(x, file) Write a comma delimited file.
write_csv2(x, file) Write a semicolon delimited file.
write_tsv(x, file) Write a tab delimited file.

One of the first steps of a project is to import outside data into R. Data is often stored in tabular formats, like csv files or spreadsheets.



The front page of this sheet shows how to import and save text files into R using **readr**.



The back page shows how to import spreadsheet data from Excel files using **readxl** or Google Sheets using **googlesheets4**.

OTHER TYPES OF DATA

Try one of the following packages to import other types of files:

- **haven** - SPSS, Stata, and SAS files
- **DBI** - databases
- **jsonlite** - json
- **xml2** - XML
- **httr** - Web APIs
- **rvest** - HTML (Web Scraping)
- **readr::read_lines()** - text data

Column Specification with readr

Column specifications define what data type each column of a file will be imported as. By default readr will generate a column spec when a file is read and output a summary.

spec(x) Extract the full column specification for the given imported data frame.

```
spec(x)
# cols(
#   age = col_integer(),
#   sex = col_character(),
#   earn = col_double()
# )
```

age is an integer
sex is a character
earn is a double (numeric)

COLUMN TYPES

Each column type has a function and corresponding string abbreviation.

- **col_logical()** - "l"
- **col_integer()** - "i"
- **col_double()** - "d"
- **col_number()** - "n"
- **col_character()** - "c"
- **col_factor(levels, ordered = FALSE)** - "f"
- **col_datetime(format = "")** - "T"
- **col_date(format = "")** - "D"
- **col_time(format = "")** - "t"
- **col_skip() - "-", "_"**
- **col_guess() - "?"**

USEFUL COLUMN ARGUMENTS

Hide col spec message

```
read_*(file, show_col_types = FALSE)
```

Select columns to import

Use names, position, or selection helpers.
read_*(file, col_select = c(age, earn))

Guess column types

To guess a column type, read_ *() looks at the first 1000 rows of data. Increase with **guess_max**.
read_*(file, guess_max = Inf)

DEFINE COLUMN SPECIFICATION

Set a default type

```
read_csv(
  file,
  col_type = list(default = col_double())
)
```

Use column type or string abbreviation

```
read_csv(
  file,
  col_type = list(x = col_double(), y = "l", z = "_")
)
```

Use a single string of abbreviations

```
# col types: skip, guess, integer, logical, character
read_csv(
  file,
  col_type = "_?ilc"
)
```

READ EXCEL FILES

A	B	C	D	E
1	x1	x2	x3	x4
2	x	z	8	
3	y	7	9	10

s1

```
read_excel(path, sheet = NULL, range = NULL)
Read a .xls or .xlsx file based on the file extension.
See front page for more read arguments. Also
read_xls() and read_xlsx().
read_excel("excel_file.xlsx")
```

READ SHEETS

A	B	C	D	E

s1 s2 s3

s1	s2	s3
----	----	----

A	B	C	D	E
A	B	C	D	E
A	B	C	D	E

s1
s1 s2 s3

```
path <- "your_file_path.xlsx"
path %>% excel_sheets() %>%
  set_names() %>%
  map_dfr(read_excel, path = path)
```

OTHER USEFUL EXCEL PACKAGES

For functions to write data to Excel files, see:

- **openxlsx**
- **writexl**

For working with non-tabular Excel data, see:

- **tidyxl**



READ SHEETS

A	B	C	D	E
1	x1	x2	x3	x4
2	x	z	8	
3	y	7	9	10

s1

```
read_sheet(ss, sheet = NULL, range = NULL)
Read a sheet from a URL, a Sheet ID, or a dribble
from the googledrive package. See front page for
more read arguments. Same as range_read().
```

SHEETS METADATA

URLs are in the form:

<https://docs.google.com/spreadsheets/d/>
SPREADSHEET_ID/edit#gid=**SHEET_ID**

gs4_get(ss) Get spreadsheet meta data.

gs4_find(...) Get data on all spreadsheet files.

sheet_properties(ss) Get a tibble of properties
for each worksheet. Also **sheet_names()**.

WRITE SHEETS

A	B	C
1	x	4
2	y	5
3	z	6

s1

write_sheet(data, ss = NULL, sheet = NULL)

Write a data frame into a new or existing Sheet.

A	B	C	D
1			
2			

s1

A	B	C
1	x1	x2
2	y	5
3	z	6

s1

gs4_create(name, ..., sheets = NULL)

Create a new Sheet with a vector of names, a data frame, or a (named) list of data frames.

A	B	C
1	x1	x2
2	1	x
3	2	y
4	3	z

s1

sheet_append(ss, data, sheet = 1)

Add rows to the end of a worksheet.

GOOGLESHEETS4 COLUMN SPECIFICATION

Column specifications define what data type each column of a file will be imported as.

Use the **col_types** argument of **read_sheet()**/**range_read()** to set the column specification.

Guess column types

To guess a column type, **read_excel()** looks at the first 1000 rows of data. Increase with the **guess_max** argument.

read_excel(path, guess_max = Inf)

Set all columns to same type, e.g. character

read_excel(path, col_types = "text")

Set each column individually

col types: skip, guess, integer, logical, character
read_sheets(ss, col_types = "_?lc")

COLUMN TYPES

I	n	c	D	L
TRUE	2	hello	1947-01-08	hello
FALSE	3.45	world	1956-10-21	1

- skip - "_" or "-"
- guess - "?"
- logical - "l"
- integer - "i"
- double - "d"
- numeric - "n"
- date - "D"
- datetime - "T"
- character - "c"
- list-column - "L"
- cell - "C" Returns list of raw cell data.

Use list for columns that include multiple data types. See **tidyxl** and **purrr** for list-column data.

FILE LEVEL OPERATIONS

googlesheets4 also offers ways to modify other aspects of Sheets (e.g. freeze rows, set column width, manage (work)sheets). Go to googlesheets4.tidyverse.org to read more.

For whole-file operations (e.g. renaming, sharing, placing within a folder), see the tidyverse package **googledrive** at googledrive.tidyverse.org.





dplyr functions work with pipes and expect **tidy data**. In tidy data:



Each **variable** is in its own **column**



Each **observation**, or **case**, is in its own **row**



`x %>% f(y)` becomes `f(x, y)`

Summarise Cases

Apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).



`summarise(.data, ...)`
Compute table of summaries.
`summarise(mtcars, avg = mean(mpg))`

`count(.data, ..., wt = NULL, sort = FALSE, name = NULL)`
Count number of rows in each group defined by the variables in ... Also **tally()**.
`count(mtcars, cyl)`

Group Cases

Use **group_by(.data, ..., .add = FALSE, .drop = TRUE)** to create a "grouped" copy of a table grouped by columns in ... dplyr functions will manipulate each "group" separately and combine the results.

`mtcars %>%
 group_by(cyl) %>%
 summarise(avg = mean(mpg))`

Use **rowwise(.data, ...)** to group data into individual rows. dplyr functions will compute results for each row. Also apply functions to list-columns. See tidyverse cheat sheet for list-column workflow.

`starwars %>%
 rowwise() %>%
 mutate(film_count = length(films))`

ungroup(x, ...) Returns ungrouped copy of table.
`ungroup(g_mtcars)`

Manipulate Cases

EXTRACT CASES

Row functions return a subset of rows as a new table.

`filter(.data, ..., .preserve = FALSE)` Extract rows that meet logical criteria.
`filter(mtcars, mpg > 20)`

`distinct(.data, ..., .keep_all = FALSE)` Remove rows with duplicate values.
`distinct(mtcars, gear)`

`slice(.data, ..., .preserve = FALSE)` Select rows by position.
`slice(mtcars, 10:15)`

`slice_sample(.data, ..., n, prop, weight_by = NULL, replace = FALSE)` Randomly select rows. Use n to select a number of rows and prop to select a fraction of rows.
`slice_sample(mtcars, n = 5, replace = TRUE)`

`slice_min(.data, order_by, ..., n, prop, with_ties = TRUE)` and `slice_max()` Select rows with the lowest and highest values.
`slice_min(mtcars, mpg, prop = 0.25)`

`slice_head(.data, ..., n, prop)` and `slice_tail()` Select the first or last rows.
`slice_head(mtcars, n = 5)`

Logical and boolean operators to use with filter()

<code>==</code>	<code><</code>	<code><=</code>	<code>is.na()</code>	<code>%in%</code>	<code> </code>	<code>xor()</code>
<code>!=</code>	<code>></code>	<code>>=</code>	<code>!is.na()</code>	<code>!</code>	<code>&</code>	

See [?base::Logic](#) and [?Comparison](#) for help.

ARRANGE CASES

`arrange(.data, ..., .by_group = FALSE)` Order rows by values of a column or columns (low to high), use with `desc()` to order from high to low.
`arrange(mtcars, mpg)`
`arrange(mtcars, desc(mpg))`

ADD CASES

`add_row(.data, ..., .before = NULL, .after = NULL)` Add one or more rows to a table.
`add_row(cars, speed = 1, dist = 1)`

Manipulate Variables

EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.

`pull(.data, var = -1, name = NULL, ...)` Extract column values as a vector, by name or index.
`pull(mtcars, wt)`

`select(.data, ...)` Extract columns as a table.
`select(mtcars, mpg, wt)`

`relocate(.data, ..., .before = NULL, .after = NULL)` Move columns to new position.
`relocate(mtcars, mpg, cyl, .after = last_col())`

Use these helpers with select() and across()

e.g. `select(mtcars, mpg:cyl)`

<code>contains(match)</code>	<code>num_range(prefix, range)</code>	<code>:</code>	e.g. <code>mpg:cyl</code>
<code>ends_with(match)</code>	<code>all_of(x)/any_of(x, ...)</code>	<code>,</code>	e.g. <code>-gear</code>
<code>starts_with(match)</code>	<code>matches(match)</code>		<code>everything()</code>

MANIPULATE MULTIPLE VARIABLES AT ONCE

`across(.cols, .funs, ..., .names = NULL)` Summarise or mutate multiple columns in the same way.
`summarise(mtcars, across(everything(), mean))`

`c_across(.cols)` Compute across columns in row-wise data.
`transmute(rowwise(UKgas), total = sum(c_across(1:2)))`

MAKE NEW VARIABLES

Apply **vectorized functions** to columns. Vectorized functions take vectors as input and return vectors of the same length as output (see back).



`mutate(.data, ..., .keep = "all", .before = NULL, .after = NULL)` Compute new column(s). Also `add_column()`, `add_count()`, and `add_tally()`.
`mutate(mtcars, gpm = 1 / mpg)`

`transmute(.data, ...)` Compute new column(s), drop others.
`transmute(mtcars, gpm = 1 / mpg)`

`rename(.data, ...)` Rename columns. Use `rename_with()` to rename with a function.
`rename(cars, distance = dist)`



Vectorized Functions

TO USE WITH MUTATE ()

mutate() and **transmute()** apply vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.

vectorized function

OFFSET

dplyr::**lag()** - offset elements by 1
dplyr::**lead()** - offset elements by -1

CUMULATIVE AGGREGATE

dplyr::**cumall()** - cumulative all()
dplyr::**cumany()** - cumulative any()
cummax() - cumulative max()
dplyr::**cummean()** - cumulative mean()
cummin() - cumulative min()
cumprod() - cumulative prod()
cumsum() - cumulative sum()

RANKING

dplyr::**cume_dist()** - proportion of all values <= dplyr::**dense_rank()** - rank w ties = min, no gaps dplyr::**min_rank()** - rank with ties = min dplyr::**ntile()** - bins into n bins dplyr::**percent_rank()** - min_rank scaled to [0,1] dplyr::**row_number()** - rank with ties = "first"

MATH

+, -, *, /, ^, %/%, %% - arithmetic ops
log(), log2(), log10() - logs
<, <=, >, >=, !=, == - logical comparisons
dplyr::**between()** - x >= left & x <= right
dplyr::**near()** - safe == for floating point numbers

MISCELLANEOUS

dplyr::**case_when()** - multi-case if_else()
starwars %>%
 mutate(type = case_when(
 height > 200 | mass > 200 ~ "large",
 species == "Droid" ~ "robot",
 TRUE ~ "other")
)
dplyr::**coalesce()** - first non-NA values by element across a set of vectors
dplyr::**if_else()** - element-wise if() + else()
dplyr::**na_if()** - replace specific values with NA
 pmax() - element-wise max()
 pmin() - element-wise min()

Summary Functions

TO USE WITH SUMMARISE ()

summarise() applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

summary function

COUNT

dplyr::**n()** - number of values/rows
dplyr::**n_distinct()** - # of uniques
sum(!is.na()) - # of non-NA's

POSITION

mean() - mean, also **mean(!is.na())**
median() - median

LOGICAL

mean() - proportion of TRUE's
sum() - # of TRUE's

ORDER

dplyr::**first()** - first value
dplyr::**last()** - last value
dplyr::**nth()** - value in nth location of vector

RANK

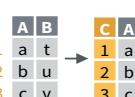
quantile() - nth quantile
min() - minimum value
max() - maximum value

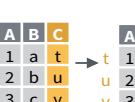
SPREAD

IQR() - Inter-Quartile Range
mad() - median absolute deviation
sd() - standard deviation
var() - variance

Row Names

Tidy data does not use rownames, which store a variable outside of the columns. To work with the rownames, first move them into a column.

 **tibble::rownames_to_column()**
Move row names into col.
a <- rownames_to_column(mtcars, var = "C")

 **tibble::column_to_rownames()**
Move col into row names.
column_to_rownames(a, var = "C")

Also **tibble::has_rownames()** and **tibble::remove_rownames()**.

Combining Tables

COMBINE VARIABLES

x	y	=
A B C a t 1 b u 2 c v 3	E F G a t 3 b u 2 d w 1	A B C E F G a t 1 a t 3 b u 2 b u 2 c v 3 d w 1

bind_cols(..., .name_repair) Returns tables placed side by side as a single table. Column lengths must be equal. Columns will NOT be matched by id (to do that look at Relational Data below), so be sure to check that both tables are ordered the way you want before binding.

RELATIONAL DATA

Use a "**Mutating Join**" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.

A B C D a t 1 3 b u 2 2 c v 3 NA	left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na")
---	---

A B C D a t 1 3 b u 2 2 d w NA 1	right_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na")
---	--

A B C D a t 1 3 b u 2 2	inner_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na")
-------------------------------	--

A B C D a t 1 3 b u 2 2 c v 3 NA d w NA 1	full_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na")
---	---

COLUMN MATCHING FOR JOINS

A B x C B y D a t 1 t 3 b u 2 u 2 c v 3 NA NA
--

Use **by = c("col1", "col2", ...)** to specify one or more common columns to match on.
left_join(x, y, by = "A")

A x B x C A y B y D a t 1 d w b u 2 b u c v 3 a t
--

Use a named vector, **by = c("col1" = "col2")**, to match on columns that have different names in each table.
left_join(x, y, by = c("C" = "D"))

A1 B1 C A2 B2 a t 1 d w b u 2 b u c v 3 a t
--

Use **suffix** to specify the suffix to give to unmatched columns that have the same name in both tables.
left_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))

COMBINE CASES

x	y	=
A B C a t 1 b u 2 c v 3	A B C c v 3 d w 4	DF A B C x a t 1 y b u 2 z c v 3 w d w 4

bind_rows(..., .id = NULL)
Returns tables one on top of the other as a single table. Set .id to a column name to add a column of the original table names (as pictured).

Use a "**Filtering Join**" to filter one table against the rows of another.

x	y	=
A B C a t 1 b u 2 c v 3	A B D a t 3 b u 2 d w 1	

semi_join(x, y, by = NULL, copy = FALSE, ..., na_matches = "na")
Return rows of x that have a match in y. Use to see what will be included in a join.

anti_join(x, y, by = NULL, copy = FALSE, ..., na_matches = "na")
Return rows of x that do not have a match in y. Use to see what will not be included in a join.

Use a "**Nest Join**" to inner join one table to another into a nested data frame.

A B C a t 1 b u 2 c v 3	y
	<tibble [1x2]> [1] a t 1 [2] b u 2 [3] c v 3

SET OPERATIONS

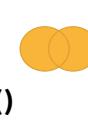
intersect(x, y, ...)
Rows that appear in both x and y.



setdiff(x, y, ...)
Rows that appear in x but not y.



union(x, y, ...)
Rows that appear in x or y. (Duplicates removed). **union_all()** retains duplicates.



Use **setequal()** to test whether two data sets contain the exact same rows (in any order).

Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same components: a **data** set, a **coordinate system**, and **geoms**—visual marks that represent data points.



To display values, map variables in the data to visual properties of the geom (**aesthetics**) like **size**, **color**, and **x** and **y** locations.



Complete the template below to build a graph.

```
ggplot (data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>),
  stat = <STAT>, position = <POSITION>) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION> +
  <SCALE_FUNCTION> +
  <THEME_FUNCTION>
```

required
Not required, sensible defaults supplied

`ggplot(data = mpg, aes(x = cty, y = hwy))` Begins a plot that you finish by adding layers to. Add one geom function per layer.

`last_plot()` Returns the last plot.

`ggsave("plot.png", width = 5, height = 5)` Saves last plot as 5' x 5' file named "plot.png" in working directory. Matches file type to file extension.

Aes Common aesthetic values.

color and **fill** - string ("red", "#RRGGBB")

linetype - integer or string (0 = "blank", 1 = "solid", 2 = "dashed", 3 = "dotted", 4 = "dotdash", 5 = "longdash", 6 = "twodash")

lineend - string ("round", "butt", or "square")

linejoin - string ("round", "mitre", or "bevel")

size - integer (line width in mm)

shape - integer/shape name or a single character ("a")


Geoms

Use a geom function to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.

GRAPHICAL PRIMITIVES

```
a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))
```

-  **a + geom_blank()** and **a + expand_limits()**
Ensure limits include values across all plots.
-  **b + geom_curve(aes(yend = lat + 1, xend = long + 1, curvature = 1))** - x, yend, alpha, angle, curvature, linetype, size
-  **a + geom_path(lineend = "butt", linejoin = "round", linemitre = 1)** - x, y, alpha, color, group, linetype, size
-  **a + geom_polygon(aes(alpha = 50))** - x, y, alpha, color, fill, group, subgroup, linetype, size
-  **b + geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1))** - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size
-  **a + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900))** - x, ymax, ymin, alpha, color, fill, group, linetype, size

LINE SEGMENTS

common aesthetics: x, y, alpha, color, linetype, size

-  **b + geom_abline(aes(intercept = 0, slope = 1))**
-  **b + geom_hline(aes(yintercept = lat))**
-  **b + geom_vline(aes(xintercept = long))**
-  **b + geom_segment(aes(yend = lat + 1, xend = long + 1))**
-  **b + geom_spoke(aes(angle = 1:1155, radius = 1))**

ONE VARIABLE continuous

- ```
c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)
```
-  **c + geom\_area(stat = "bin")** - x, y, alpha, color, fill, linetype, size
  -  **c + geom\_density(kernel = "gaussian")** - x, y, alpha, color, fill, group, linetype, size, weight
  -  **c + geom\_dotplot()** - x, y, alpha, color, fill
  -  **c + geom\_freqpoly()** - x, y, alpha, color, group, linetype, size
  -  **c + geom\_histogram(binwidth = 5)** - x, y, alpha, color, fill, linetype, size, weight
  -  **c2 + geom\_qq(aes(sample = hwy))** - x, y, alpha, color, fill, linetype, size, weight

## discrete

```
d <- ggplot(mpg, aes(fl))
```

-  **d + geom\_bar()** - x, alpha, color, fill, linetype, size, weight

## TWO VARIABLES both continuous

```
e <- ggplot(mpg, aes(cty, hwy))
```

-  **e + geom\_label(aes(label = cty), nudge\_x = 1, nudge\_y = 1)** - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust
-  **e + geom\_point()** - x, y, alpha, color, fill, shape, size, stroke
-  **e + geom\_quantile()** - x, y, alpha, color, group, linetype, size, weight
-  **e + geom\_rug(sides = "bl")** - x, y, alpha, color, linetype, size
-  **e + geom\_smooth(method = lm)** - x, y, alpha, color, fill, group, linetype, size, weight
-  **e + geom\_text(aes(label = cty), nudge\_x = 1, nudge\_y = 1)** - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

## one discrete, one continuous

```
f <- ggplot(mpg, aes(class, hwy))
```

-  **f + geom\_col()** - x, y, alpha, color, fill, group, linetype, size
-  **f + geom\_boxplot()** - x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight
-  **f + geom\_dotplot(binaxis = "y", stackdir = "center")** - x, y, alpha, color, fill, group
-  **f + geom\_violin(scale = "area")** - x, y, alpha, color, fill, group, linetype, size, weight

## both discrete

```
g <- ggplot(diamonds, aes(cut, color))
```

-  **g + geom\_count()** - x, y, alpha, color, fill, shape, size, stroke
-  **e + geom\_jitter(height = 2, width = 2)** - x, y, alpha, color, fill, shape, size

## THREE VARIABLES

```
seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2)); l <- ggplot(seals, aes(long, lat))
```

-  **l + geom\_contour(aes(z = z))** - x, y, z, alpha, color, group, linetype, size, weight
-  **l + geom\_contour\_filled(aes(fill = z))** - x, y, alpha, color, fill, group, linetype, size, subgroup

## continuous bivariate distribution

```
h <- ggplot(diamonds, aes(carat, price))
```

-  **h + geom\_bin2d(binwidth = c(0.25, 500))** - x, y, alpha, color, fill, linetype, size, weight
-  **h + geom\_density\_2d()** - x, y, alpha, color, group, linetype, size
-  **h + geom\_hex()** - x, y, alpha, color, fill, size

## continuous function

```
i <- ggplot(economics, aes(date, unemploy))
```

-  **i + geom\_area()** - x, y, alpha, color, fill, linetype, size
-  **i + geom\_line()** - x, y, alpha, color, group, linetype, size
-  **i + geom\_step(direction = "hv")** - x, y, alpha, color, group, linetype, size

## visualizing error

```
df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
j <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))
```

-  **j + geom\_crossbar(fatten = 2)** - x, y, ymax, ymin, alpha, color, fill, group, linetype, size
-  **j + geom\_errorbar()** - x, y, ymax, ymin, alpha, color, group, linetype, size, width  
Also **geom\_errorbarh()**.
-  **j + geom\_linerange()** - x, ymin, ymax, alpha, color, group, linetype, size
-  **j + geom\_pointrange()** - x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

## maps

```
data <- data.frame(murder = USARests$Murder,
state = tolower(rownames(USARests)))
map <- map_data("state")
k <- ggplot(data, aes(fill = murder))
```

-  **k + geom\_map(aes(map\_id = state), map = map) + expand\_limits(x = map\$long, y = map\$lat)** - map\_id, alpha, color, fill, linetype, size

-  **l + geom\_raster(aes(fill = z), hjust = 0.5, vjust = 0.5, interpolate = FALSE)** - x, y, alpha, fill
-  **l + geom\_tile(aes(fill = z))** - x, y, alpha, color, fill, linetype, size, width



## Basics

`data.table` is an extremely fast and memory efficient package for transforming data in R. It works by converting R's native data frame objects into `data.tables` with new and enhanced functionality. The basics of working with `data.tables` are:

**`dt[i, j, by]`**

Take `data.table dt`,  
subset rows using `i`  
and manipulate columns with `j`,  
grouped according to `by`.

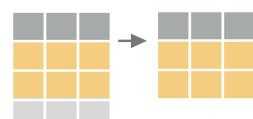
`data.tables` are also data frames – functions that work with data frames therefore also work with `data.tables`.

## Create a `data.table`

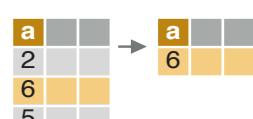
`data.table(a = c(1, 2), b = c("a", "b"))` – create a `data.table` from scratch. Analogous to `data.frame()`.

`setDT(df)*` or `as.data.table(df)` – convert a data frame or a list to a `data.table`.

## Subset rows using `i`



`dt[1:2, ]` – subset rows based on row numbers.



`dt[a > 5, ]` – subset rows based on values in one or more columns.

### LOGICAL OPERATORS TO USE IN `i`

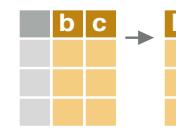
|   |    |          |      |   |           |
|---|----|----------|------|---|-----------|
| < | <= | is.na()  | %in% |   | %like%    |
| > | >= | !is.na() | !    | & | %between% |

## Manipulate columns with `j`

### EXTRACT



`dt[, c(2)]` – extract columns by number. Prefix column numbers with “-” to drop.



`dt[, .(b, c)]` – extract columns by name.

### SUMMARIZE



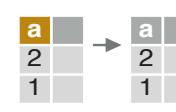
`dt[, .(x = sum(a))]` – create a `data.table` with new columns based on the summarized values of rows.

Summary functions like `mean()`, `median()`, `min()`, `max()`, etc. can be used to summarize rows.

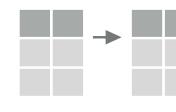
### COMPUTE COLUMNS\*



`dt[, c := 1 + 2]` – compute a column based on an expression.



`dt[a == 1, c := 1 + 2]` – compute a column based on an expression but only for a subset of rows.



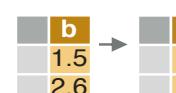
`dt[, `:=` (c = 1, d = 2)]` – compute multiple columns based on separate expressions.

### DELETE COLUMN



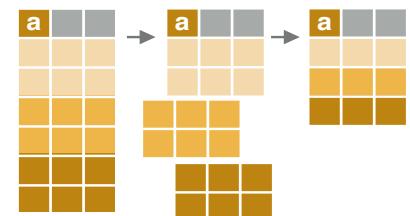
`dt[, c := NULL]` – delete a column.

### CONVERT COLUMN TYPE



`dt[, b := as.integer(b)]` – convert the type of a column using `as.integer()`, `as.numeric()`, `as.character()`, `as.Date()`, etc..

## Group according to `by`



`dt[, j, by = .(a)]` – group rows by values in specified columns.

`dt[, j, keyby = .(a)]` – group and simultaneously sort rows by values in specified columns.

### COMMON GROUPED OPERATIONS

`dt[, .(c = sum(b)), by = a]` – summarize rows within groups.

`dt[, c := sum(b), by = a]` – create a new column and compute rows within groups.

`dt[, .SD[1], by = a]` – extract first row of groups.

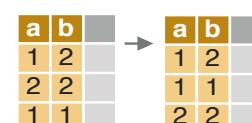
`dt[, .SD[N], by = a]` – extract last row of groups.

## Chaining

`dt[...][...]` – perform a sequence of `data.table` operations by chaining multiple “`[]`”.

## Functions for `data.tables`

### REORDER



`setorder(dt, a, -b)` – reorder a `data.table` according to specified columns. Prefix column names with “-” for descending order.

### \* SET FUNCTIONS AND `:=`

`data.table`'s functions prefixed with “`set`” and the operator “`:=`” work without “`<-`” to alter data without making copies in memory. E.g., the more efficient “`setDT(df)`” is analogous to “`df <- as.data.table(df)`”.

# Apply function to cols.

## APPLY A FUNCTION TO MULTIPLE COLUMNS

|     |   |     |
|-----|---|-----|
| a b | → | a b |
| 1 2 |   | 1 2 |
| 2 2 |   | 2 2 |
| 1 2 |   | 2 2 |

`dt[, lapply(.SD, mean), .SDcols = c("a", "b")]` – apply a function – e.g. `mean()`, `as.character()`, `which.max()` – to columns specified in `.SDcols` with `lapply()` and the `.SD` symbol. Also works with groups.

|     |   |     |
|-----|---|-----|
| a b | → | a b |
| 1 4 |   | 2 5 |
| 2 5 |   |     |
| 3 6 |   |     |

`cols <- c("a")`  
`dt[, paste0(cols, "_m") := lapply(.SD, mean), .SDcols = cols]` – apply a function to specified columns and assign the result with suffixed variable names to the original data.

# Sequential rows

## ROW IDS

|     |   |       |
|-----|---|-------|
| a b | → | a b c |
| 1 a |   | 1 a 1 |
| 2 a |   | 2 a 2 |
| 3 b |   | 3 b 1 |

`dt[, c := 1:N, by = b]` – within groups, compute a column with sequential row IDs.

## LAG & LEAD

|     |   |        |
|-----|---|--------|
| a b | → | a b c  |
| 1 a |   | 1 a NA |
| 2 a |   | 2 a 1  |
| 3 b |   | 3 b NA |
| 4 b |   | 4 b 3  |
| 5 b |   | 5 b 4  |

`dt[, c := shift(a, 1), by = b]` – within groups, duplicate a column with rows lagged by specified amount.

`dt[, c := shift(a, 1, type = "lead"), by = b]` – within groups, duplicate a column with rows leading by specified amount.

# read & write files

## IMPORT

`fread("file.csv")` – read data from a flat file such as `.csv` or `.tsv` into R.

`fread("file.csv", select = c("a", "b"))` – read specified columns from a flat file into R.

## EXPORT

`fwrite(dt, "file.csv")` – write data to a flat file from R.

## UNIQUE ROWS

|     |   |     |
|-----|---|-----|
| a b | → | a b |
| 1 2 |   | 1 2 |
| 2 2 |   | 2 2 |
| 1 2 |   | 2 2 |

`unique(dt, by = c("a", "b"))` – extract unique rows based on columns specified in “by”. Leave out “by” to use all columns.

## RENAME COLUMNS

|     |   |     |
|-----|---|-----|
| a b | → | x y |
| 1 2 |   | 1 2 |

`setnames(dt, c("a", "b"), c("x", "y"))` – rename columns.

## SET KEYS

`setkey(dt, a, b)` – set keys to enable fast repeated lookup in specified columns using “`dt[.(value), ]`” or for merging without specifying merging columns using “`dt_a[dt_b]`”.

# Combine data.tables

## JOIN

|     |   |     |   |       |
|-----|---|-----|---|-------|
| a b | + | x y | = | a b x |
| 1 c |   | 3 b |   | 3 b 3 |
| 2 a |   | 2 c |   | 1 c 2 |
| 3 b |   | 1 a |   | 2 a 1 |

`dt_a[dt_b, on = .(b = y)]` – join data.tables on rows with equal values.

|       |   |       |   |          |
|-------|---|-------|---|----------|
| a b c | + | x y z | = | a b c x  |
| 1 c 7 |   | 3 b 4 |   | 3 b 4 3  |
| 2 a 5 |   | 2 c 5 |   | 1 c 5 2  |
| 3 b 6 |   | 1 a 8 |   | NA a 8 1 |

`dt_a[dt_b, on = .(b = y, c > z)]` – join data.tables on rows with equal and unequal values.

## ROLLING JOIN

|                |   |                |   |                    |
|----------------|---|----------------|---|--------------------|
| a id date      | + | b id date      | = | a id date   b      |
| 1 A 01-01-2010 |   | 1 A 01-01-2013 |   | 2 A 01-01-2013   1 |
| 2 A 01-01-2012 |   | 1 B 01-01-2013 |   | 2 B 01-01-2013   1 |
| 3 A 01-01-2014 |   |                |   |                    |
| 1 B 01-01-2010 |   |                |   |                    |
| 2 B 01-01-2012 |   |                |   |                    |

`dt_a[dt_b, on = .(id = id, date = date), roll = TRUE]` – join data.tables on matching rows in id columns but only keep the most recent preceding match with the left data.table according to date columns. “`roll = -Inf`” reverses direction.

## BIND

|     |   |     |   |     |
|-----|---|-----|---|-----|
| a b | + | a b | = | a b |
| 1 2 |   | 1 2 |   | 1 2 |

`rbind(dt_a, dt_b)` – combine rows of two data.tables.

|     |   |     |   |         |
|-----|---|-----|---|---------|
| a b | + | x y | = | a b x y |
| 1 2 |   | 1 2 |   | 1 2 1 2 |

`cbind(dt_a, dt_b)` – combine columns of two data.tables.

# Reshape a data.table

## RESHAPE TO WIDE FORMAT

|          |   |                    |
|----------|---|--------------------|
| id y a b | → | id a x a z b x b z |
| A x 1 3  |   | A 1 2 3 4          |
| A z 2 4  |   | B 1 2 3 4          |

`dcast(dt, id ~ y, value.var = c("a", "b"))`

Reshape a data.table from long to wide format.

`dt` A data.table.  
`id ~ y` Formula with a LHS: ID columns containing IDs for multiple entries. And a RHS: columns with values to spread in column headers.  
`value.var` Columns containing values to fill into cells.

## RESHAPE TO LONG FORMAT

|                    |   |          |
|--------------------|---|----------|
| id a_x a_z b_x b_z | → | id y a b |
| A 1 2 3 4          |   | A 1 1 3  |
| B 1 2 3 4          |   | B 1 1 3  |

`melt(dt, id.vars = c("id"), measure.vars = patterns("^a", "^b"), variable.name = "y", value.name = c("a", "b"))`

Reshape a data.table from wide to long format.

`dt` A data.table.  
`id.vars` ID columns with IDs for multiple entries.  
`measure.vars` Columns containing values to fill into cells (often in pattern form).  
`variable.name` Names of new columns for variables and values derived from old headers.  
`value.name`

## Model

What is your model of the world, including how outcomes respond to interventions in the world?

### Population

Define the size of the population, hierarchical structure (if any), and background variables.

Simple dataset with no background variables

```
pop <- declare_population(N = 100)
pop()
```

Simple dataset with background variables

```
declare_population(N = 100,
 X = rnorm(N))
```

Two-level dataset

```
declare_population(
 schools =
 add_level(N = 10,
 funding = rnorm(N)),
 students =
 add_level(N = 100,
 scores = rnorm(N))
)
```

## Outcomes

### Outcomes that depend on a treatment (Z)

Using a formula

```
declare_potential_outcomes(
 Y ~ .5 * Z + rnorm(N))
```

As separate variables

```
declare_potential_outcomes(
 Y_Z_0 = rnorm(N),
 Y_Z_1 = Y_Z_0 + .5)
```

### Outcomes that do not depend on treatment

```
declare_potential_outcomes(
 Y = rnorm(N))
```

## Inquiry

What is the research question you want to answer?

Causal inquiries

```
declare_estimand(
 ATE = mean(Y_Z_1 - Y_Z_0))
```

Descriptive inquiries

```
declare_estimand(
 Y_median = median(Y))
```

Conditional estimands

```
declare_estimand(
 LATE = mean(Y_Z_1 - Y_Z_0),
 subset = complier == TRUE)
```

## Data Strategy

How will you generate data to answer your inquiry?

### Sampling

```
declare_sampling(n = 100)
```

```
declare_sampling(
 strata_n = 20,
 strata = urban_area)
```

### Treatment assignment

```
declare_assignment(m = 100)
```

```
declare_assignment(
 clusters = villages,
 m = 10)
```

## Answer Strategy

How will you generate an answer to your inquiry?

OLS with robust standard errors

```
declare_estimator(
 Y ~ Z, model = lm_robust)
```

2SLS instrumental variables regression with robust SEs

```
declare_estimator(
 Y ~ D | Z, model = iv_robust)
```

Difference-in-means

```
declare_estimator(
 Y ~ Z,
 model = difference_in_means)
```

**DeclareDesign** is a software implementation of the MIDA framework, according to which research designs have a **Model** of the world, an **Inquiry** about that model, a **Data strategy** that generates information about the world, and an **Answer** strategy that uses data to make a guess about the **Inquiry**. Declared designs can be “diagnosed” to calculate the properties of the design such as power and bias using Monte Carlo simulation.

All `declare_*` functions return *functions*. Most functions take a `data.frame` and return a `data.frame`.

## Design Declaration

Put together all the steps into a declared design using the `+` operator

```
design <-
 declare_population(N = 200, X = rnorm(N)) +
 declare_potential_outcomes(Y ~ .5 * Z + X) +
 declare_estimand(ATR = mean(Y_Z_1 - Y_Z_0)) +
 declare_sampling(n = 100) +
 declare_assignment(m = 50) +
 declare_estimator(Y ~ Z, model = lm_robust)
```

```
draw_data(design)
draw_estimates(design)
get_estimates(design, data = real_data)
draw_estimands(design)
run_design(design)
summary(design)
compare_designs(design_1, design_2)
```

## Design Diagnosis

Diagnose the properties of your design

```
diagnosis <- diagnose_design(
 design, sims = 100, bootstrap_sims = 100)
```

```
summary(diagnosis)
get_diagnosands(diagnosis)
get_simulations(diagnosis)
```

Custom diagnosands

```
diagnose_design(
 design,
 diagnosands = declare_diagnosands(
 sig_pos = mean(p.value < .05 & estimate > 0)))
```

## Introduction

distr6 is an object-oriented interface for probability distributions. Including distributions as objects, statistical properties of distributions, composite modelling and decorators for numerical imputation. As well as this cheat sheet, see:

- [GitHub](#) for an issue tracker and latest development branch
- [CRAN](#) for package meta-data
- The distr6 [website](#) for more complete tutorials.

## R6 Classes

|                      |                                                                     |                                                              |
|----------------------|---------------------------------------------------------------------|--------------------------------------------------------------|
| <b>Distribution</b>  | The parent class to most distr6 classes.                            | <code>Distribution</code>                                    |
| <b>SDistribution</b> | Class given to all probability distributions implemented in distr6. | <code>Distribution</code><br>↑<br><code>SDistribution</code> |
| <b>Kernel</b>        | Class given to all kernel-like probability distributions.           | <code>Distribution</code><br>↑<br><code>Kernel</code>        |
| <b>Decorator</b>     | Used to add or impute methods to a Distribution.                    |                                                              |
| <b>Wrapper</b>       | Create composite distributions by adapting class properties         |                                                              |
| <b>ParameterSet</b>  | Class used to add parameters to a distribution.                     |                                                              |

## R6 Basics

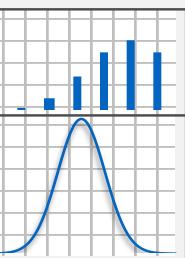
|                                                        |                                                                                        |
|--------------------------------------------------------|----------------------------------------------------------------------------------------|
| \$ All methods are called using dollar-sign notation   | <code>N &lt;- Normal\$new()</code><br><code>N\$mean()</code><br><code>N\$pdf(2)</code> |
| <b>clone</b> Objects are copied using the clone method | <code>N1 &lt;- Normal\$new()</code><br><code>N2 &lt;- N1\$clone()</code>               |
| <b>Method chaining</b> Call one method after another   | <code>Normal\$new()\$pdf(2)</code>                                                     |

## Construct a Distribution

Each distribution has a default parameterisation, and all common parameterisations are available.

```
Binomial$new()
Binomial$new(size=5, prob=0.6)
Binomial$new(size=5, qprob=0.4)

Normal$new()
Normal$new(mean=0, sd=1)
Normal$new(mean=0, var=1)
Normal$new(mean=0, prec=1)
```



You can list all the implemented probability distributions and kernels

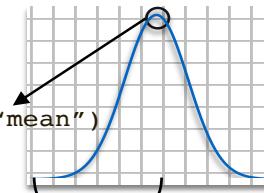
```
listDistributions()
listKernels()
```

## Get and Set Parameters

```
N <- Normal$new()
```

```
N$parameters()
```

```
N$getParameterValue("mean")
```



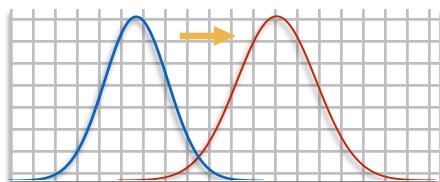
```
N$getParameterValue("variance")
```

Any parameter can be set, even if it wasn't used in construction. And multiple can be updated at the same time.

```
N$setParameterValue(mean = 2)
```

```
N$setParameterValue(prec = 2)
```

```
N$setParameterValue(mean = 3, sd = 3)
```



## Statistical Methods

```
N <- Normal$new()
```

```
N$mean()
```

```
N$variance()
```

```
N$skewness()
```

```
N$kurtosis()
```

```
N$entropy()
```

Use ?SDistribution, ?Normal (or any other distribution) to see available methods.

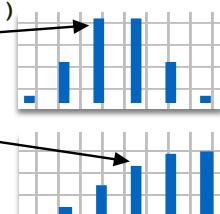
```
B <- Binomial$new(size = 5)
```

```
B$pdf(0:5)
```

```
B$cdf(0:5)
```

```
B$quantile(0.42)
```

```
B$rand(5)
```



Once again vectorization is available

```
> MN <- MultivariateNormal$new(mean = c(0,0,0), cov = c(3,-1,-1,-1,1,0,-1,0,1))

> MN <- MultivariateNormal$new(mean = c(0,0,0), prec = c(3,-1,-1,-1,1,0,-1,0,1))

> MN$pdf(1, 2, 3)
> MN$cdf(1, 1, 1)
```

```
> MN$pdf(1:2, 2:3, 1:2)
> MN$cdf(c(0.45, 0.65),
 c(0.12, 0.99), c(0, 1))
```

## S3 and Piping

distr6 uses '[R62S3](#)' so every R6 method has an S3 dispatch available.

```
N <- Normal$new()

N$mean() ➔ mean(N)

N$getParameterValue("mean") ➔ getParameterValue(N, "mean")

N$pdf(1:5) ➔ pdf(N, 1:5)
```

Use the 'magrittr' package for method chaining and piping (%>%).

```
> N <- Normal$new()
> N$setParameterValue(sd=2)$getParameterValue("var")
library(magrittr)

> N <- Normal$new()
> N %>% setParameterValue(sd=2) %>%
 getParameterValue("var")
```

## Multivariate Distributions

Multivariate distributions are handled just like univariate distributions, except the pdf/cdf functions take multiple arguments, as do cf and mgf where available.

```
> MN <- MultivariateNormal$new(mean = c(0,0,0), cov = c(3,-1,-1,-1,1,0,-1,0,1))

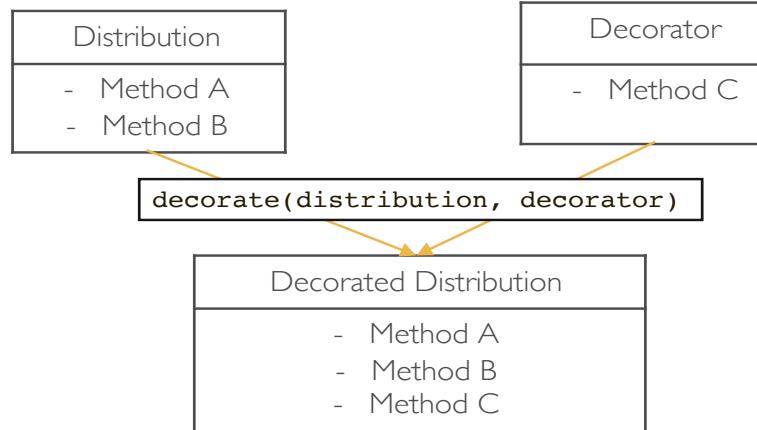
> MN <- MultivariateNormal$new(mean = c(0,0,0), prec = c(3,-1,-1,-1,1,0,-1,0,1))

> MN$pdf(1, 2, 3)
> MN$cdf(1, 1, 1)
```

```
> MN$pdf(1:2, 2:3, 1:2)
> MN$cdf(c(0.45, 0.65),
 c(0.12, 0.99), c(0, 1))
```

## Decorators

Decorators are a design pattern (Gamma et al., 1994) used to add methods to objects.



## Available Decorators

**CoreStatistics** Imputes common numeric statistical results, adds generalised expectation and moments function.

**ExoticStatistics** Adds methods for survival analysis and statistical modelling.

**FunctionImputation** Uses numerical methods to impute missing pdf/cdf/quantile/rand functions

Remember to decorate first before using a method from a decorator

```
> N <- Normal$new()
> N$survival(1)
Error: attempt to apply non-function
> decorate(N, ExoticStatistics)
> N$survival(1)
[1] 0.1586553
```

S3 methods will now work too

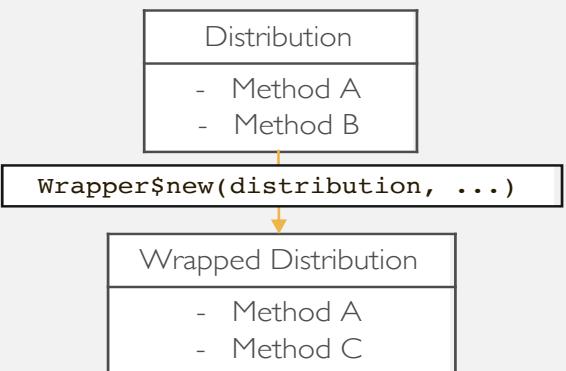
```
> N <- Normal$new(decorators = ExoticStatistics)
> pdfPNorm(N, 3, -1, 1)
[1] 0.4383636
```

Use listing to see which decorators are currently implemented.

```
listDecorators()
```

## Wrappers

Wrappers are based on the **Adapter** design pattern (Gamma et al., 1994) and are used to change the interface of an object.



## Available Wrappers

### ProductDistribution

Product of two or more distributions.

### Convolution

Addition (or subtraction) of two distributions

### HuberizedDistribution

Huberizes a distribution between limits.

### VectorDistribution

Vectorizes two or more distributions.

### MixtureDistribution

Weighted mixture of two or more distributions

### TruncatedDistribution

Truncates a distribution between limits.



```
> TruncatedDistribution$new(Normal$new(),
+ lower = -1, upper = 1)
> MixtureDistribution$new(list(Binomial$new(),
+ Normal$new()), weights = c(0.4, 0.6))
> ProductDistribution$new(list(Exponential$new(),
+ Normal$new()))$pdf(1,1)
```

Use listing to see which wrappers are currently implemented.

```
listWrappers()
```

## Custom Distributions

Custom distributions can be created using `Distribution$new`, this is not the same as implementing a new `SDistribution`!

```
pdf <-
function(x1) return(1/(self$getParameterValue("upper") - self$getParameterValue("lower")))
```

The `self` argument tells the object to call the method on itself.

All `pdf/cdf` methods in `distr6` use '`x1, x2, ...`' as their arguments

```
cdf <- function(x1) return((x1 -
self$getParameterValue("lower")) /
(self$getParameterValue("upper") -
self$getParameterValue("lower")))
```

`ParameterSet` is the class used for `distr6` parameters.

```
ps <- ParameterSet$new(id = list("lower", "upper"),
value = c(1, 10), support =
list(Reals$new(), Reals$new()), settable =
list(TRUE, TRUE))
```

The argument `support` is of type `SetInterval`.  
See `listSpecialSets()`

Unique distribution `name` and one-word `short_name` (ID)

```
dist <- Distribution$new(name = "Uniform",
short_name = "unif", type = Reals$new(), support =
Interval$new(1, 10), symmetric = TRUE, pdf = pdf,
cdf = cdf, parameters = ps, description = "Custom
uniform distribution", decorators = CoreStatistics)
```

`Distribution type` and `support` is of type `SetInterval`.

`CoreStatistics` decorator is optionally used to impute numeric results.

`log` and `lower.tail` arguments are added automatically

```
> dist$pdf(1, log = TRUE)
[1] -2.197225
> dist$cdf(2, lower.tail = FALSE)
[1] 0.8888889
> decorate(dist, FunctionImputation)
> dist$mean()
[1] 5.5
> dist$quantile(0.42)
[1] 4.78
```

impute missing `quantile` and `rand` methods

## OLS with lm\_robust()

lm\_robust() is lm() with robust SEs. HC2 is the default.

```
lm_robust(mpg ~ hp, data = mtcars)
lm_robust(mpg ~ hp, se_type = "HC1",
 data = mtcars)
lm_robust(mpg ~ hp, se_type = "classical",
 data = mtcars)
```

Indicate clusters to get clustered SEs. CR2 is the default.

```
lm_robust(mpg ~ hp, clusters = carb,
 data = mtcars)
lm_robust(mpg ~ hp, clusters = carb,
 se_type = "stata", data = mtcars)
```

**Fixed effects** two ways:

```
FEs as "dummies"
lm_robust(mpg ~ hp + as.factor(am),
 data = mtcars)

"Absorbing" FEs (substantially faster)
lm_robust(mpg ~ hp,
 fixed_effects = ~ am,
 data = mtcars)
```

**post-estimation** commands:

```
fit <- lm_robust(mpg ~ hp, data = mtcars)
summary(fit)
print(fit)
tidy(fit)
vcov(fit)
confint(fit)
nobs(fit)
predict(fit, newdata = mtcars)
```

estimatr is part of the DeclareDesign suite of packages for designing, implementing, and analyzing social science research designs.

## 2SLS with iv\_robust()

iv\_robust() is AER::ivreg() with robust SEs.

```
iv_robust(mpg ~ hp | am, data = mtcars)
iv_robust(mpg ~ hp | am,
 clusters = carb, data = mtcars)
```

## Two-group estimators

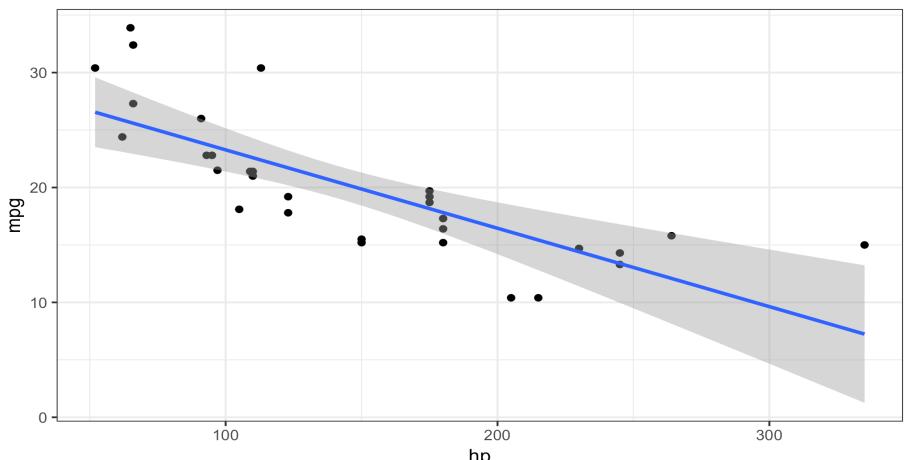
difference\_in\_means() and horvitz\_thompson()  
compare two groups

```
difference_in_means(mpg ~ am, data = mtcars)
horvitz_thompson(mpg ~ am, data = mtcars)
```

## ggplot2 integration

Use robust variance estimates for drawing confidence intervals:

```
library(ggplot2)
ggplot(mtcars, aes(mpg, hp)) +
 geom_point() +
 stat_smooth(method = "lm_robust") +
 theme_bw()
```



## Multiple models

Same outcome, different subsets:

```
library(tidyverse)
mtcars %>%
 split(. $cyl) %>%
 map(~lm_robust(mpg ~ hp, data = .)) %>%
 map(tidy) %>%
 bind_rows(. id = "cyl")
```

Different outcomes, same subset:

```
c("mpg", "disp") %>%
 map(~formula(paste0(., " ~ hp"))) %>%
 map(~lm_robust(., data = mtcars)) %>%
 map(tidy) %>%
 bind_rows
```

## Extras

```
Lin (2013) covariate adjustment
lm_lin(mpg ~ am, covariates = ~ hp,
 data = mtcars)

regression tables with texreg
fit <- lm_robust(mpg ~ hp, data = mtcars)
texreg::texreg(fit, include.ci = FALSE)
```

## estimatr-to-Stata dictionary

### estimatr

```
lm_robust(y ~ z,
 data = dat)
```

### Stata

```
reg y z, vce(hc2)
```

```
lm_robust(y ~ z,
 clusters = cl,
 se_type = "stata",
 data = dat)
```

```
reg y z, vce(cluster cl)
```

```
lm_robust(mpg ~ hp,
 fixed_effects = ~ am,
 se_type = "stata",
 data = mtcars)
```

```
areg mpg hp, absorb(am)
vce(robust)
```

```
iv_robust(mpg ~ hp | am,
 se_type = "HC1",
 data = mtcars)
```

```
ivregress 2sls mpg (hp =
am), vce(robust) small
```

# Search and download

Data in the Eurostat database is stored in tables. Each table has an identifier, a short table\_code, and a description (e.g. tps00199 - Total fertility rate).

Key eurostat functions allow to find the table\_code, download the eurostat table and polish labels in the table.

## Find the table code

The **search\_eurostat(pattern,...)** function scans the directory of Eurostat tables and returns codes and descriptions of tables that match pattern.

```
library("eurostat")
query <- search_eurostat(pattern = "fertility rate",
 type = "table", fixed = FALSE)
query[,1:2]
title code
<chr> <chr>
Total fertility rate by NUTS 2 region tgs00100
Total fertility rate tps00199
Total fertility rate by NUTS 2 region tgs00100
```

## Download the table

The **get\_eurostat(id, time\_format = "date", filters = "none", type = "code", cache = TRUE,...)** function downloads the requested table from the Eurostat bulk download facility or from The Eurostat Web Services JSON API (if filters are defined). Downloaded data is cached (if cache=TRUE). Additional arguments define how to read the time column (time\_format) and if table dimensions shall be kept as codes or converted to labels (type).

```
ct <- c("AT", "BE", "BG", "CH", "CY", "CZ", "DE", "DK", "EE", "EL", "ES",
 "FI", "FR", "HR", "HU", "IE", "IS", "IT", "LI", "LT", "LU", "LV",
 "MT", "NL", "NO", "PL", "PT", "RO", "SE", "SI", "SK", "UK")
dat <- get_eurostat(id="tps00199", time_format="num",
 filters = list(geo = ct))
dat[1:2,]
indic_de geo time values
TOTFERRT AT 2006 1.41
TOTFERRT AT 2007 1.38
```

## Add labels

The **label\_eurostat(x, lang = "en",...)** gets definitions for Eurostat codes and replace them with labels in given language ("en", "fr" or "de")

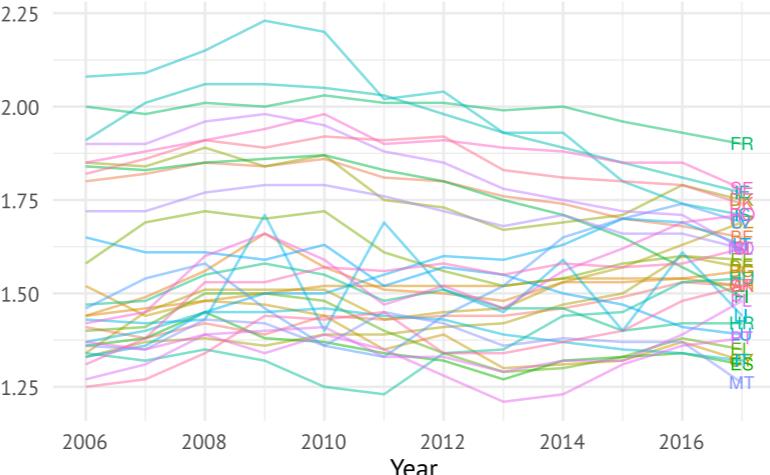
```
dat <- label_eurostat(dat)
dat[1:3,]
indic_de geo time values
<fct> <fct> <dbl> <dbl>
Total fertility rate Andorra 2006 1.24
Total fertility rate Albania 2006 1.67
Total fertility rate Armenia 2006 1.34
```

# eurostat and plots

The **get\_eurostat()** function returns tibbles in the long format. Packages dplyr and tidyr are well suited to transform these objects. The **ggplot2**-package is well suited to plot these objects.

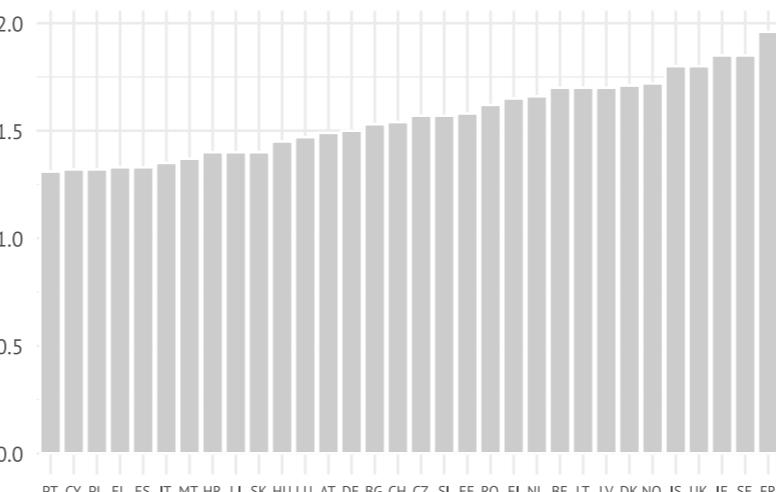
```
dat <- get_eurostat(id="tps00199", filters = list(geo = ct))
library(ggplot2)
library(dplyr)
ggplot(dat,
 aes(x = time, y = values, color = geo, label = geo)) +
 geom_line(alpha = .5) +
 geom_text(data = dat %>% group_by(geo) %>%
 filter(time == max(time)),
 size = 2.6) +
 theme(legend.position = "none") +
 labs(title = "Total fertility rate, 2006-2017",
 x = "Year", y = "%")
```

Total fertility rate, 2006-2017



```
dat_2015 <- dat %>%
 filter(time == "2015-01-01")
ggplot(dat_2015, aes(x = reorder(geo, values), y = values)) +
 geom_col(color = "white", fill = "grey80") +
 theme(axis.text.x = element_text(size = 6)) +
 labs(title = "Total fertility rate, 2015",
 y = "%", x = NULL)
```

Total fertility rate, 2015



# eurostat and maps

There are two function to work with geospatial data from eurostat. **get\_eurostat\_geospatial()** returns spatial data as sf-object. Object can me merged with data.frames using **dplyr::left\_join()**. The **cut\_to\_classes()** is a wrapper for **cut()** - function and is used for categorizing data for maps with tidy labels.

```
mapdata <- get_eurostat_geospatial(nuts_level =
 right_join(dat_2015) %>%
 mutate(cat = cut_to_classes(values, n=4, decimal = 1),
 head(select(mapdata, geo, values, cat), 3))
geo values cat
AT 1.49 1.5 ~< 1.6 MULTIPOLYGON (((15.54245...
```

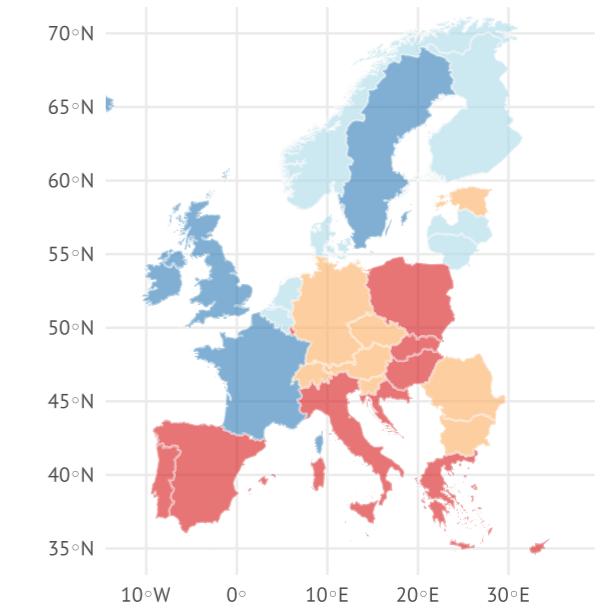
## Plot a Map

The **sf-object** returned are ready to be plotted with **ggplot::geom\_sf()**-function.

```
ggplot(mapdata, aes(fill = cat)) +
 scale_fill_brewer(palette = "RdYlBu") +
 geom_sf(color = alpha("white", 1/3), alpha = .5) +
 xlim(c(-12, 44)) + ylim(c(35, 70)) +
 labs(title = "Total fertility rate, 2015",
 subtitle = "Avg. number of life births per woman",
 fill = "%")
```

Total fertility rate, 2015

Avg. number of life births per woman



This onepager presents the eurostat package 2014-2019. Leo Lahti, Janne Huovari, Markus Kainu, Przemyslaw Biecek, package version 3.3.55 URL: <https://github.com/rOpenGov/eurostat>

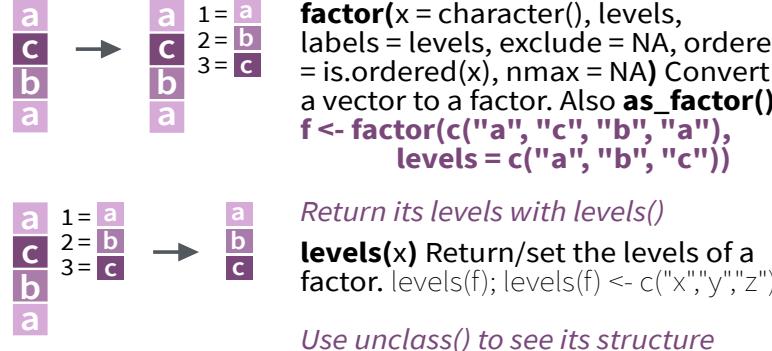
Retrieval and Analysis of Eurostat Open Data with the eurostat R package. Leo Lahti, Janne Huovari, Markus Kainu, and Przemysław Biecek. The R Journal, 9(1):385–392, 2017.



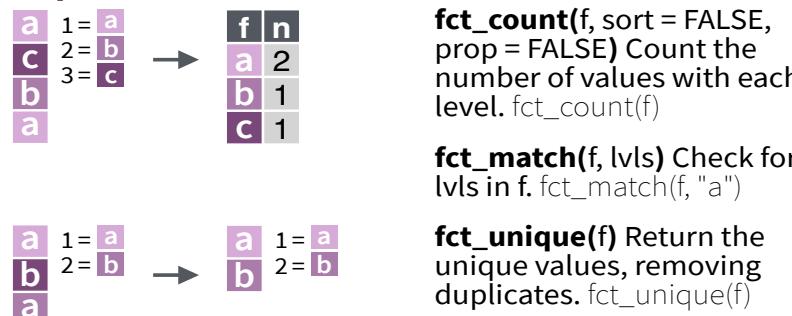
The **forcats** package provides tools for working with factors, which are R's data structure for categorical data.

## Factors

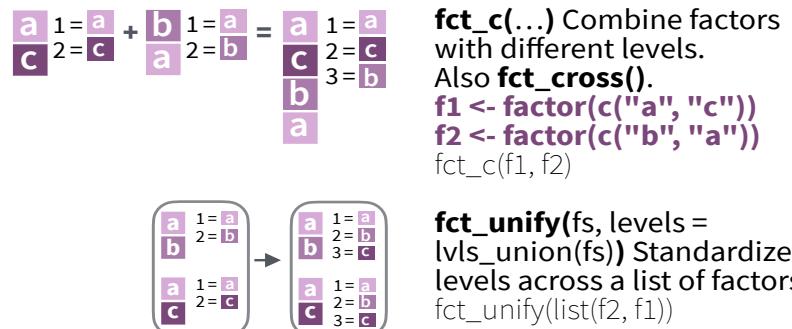
R represents categorical data with factors. A **factor** is an integer vector with a **levels** attribute that stores a set of mappings between integers and categorical values. When you view a factor, R displays not the integers, but the levels associated with them.



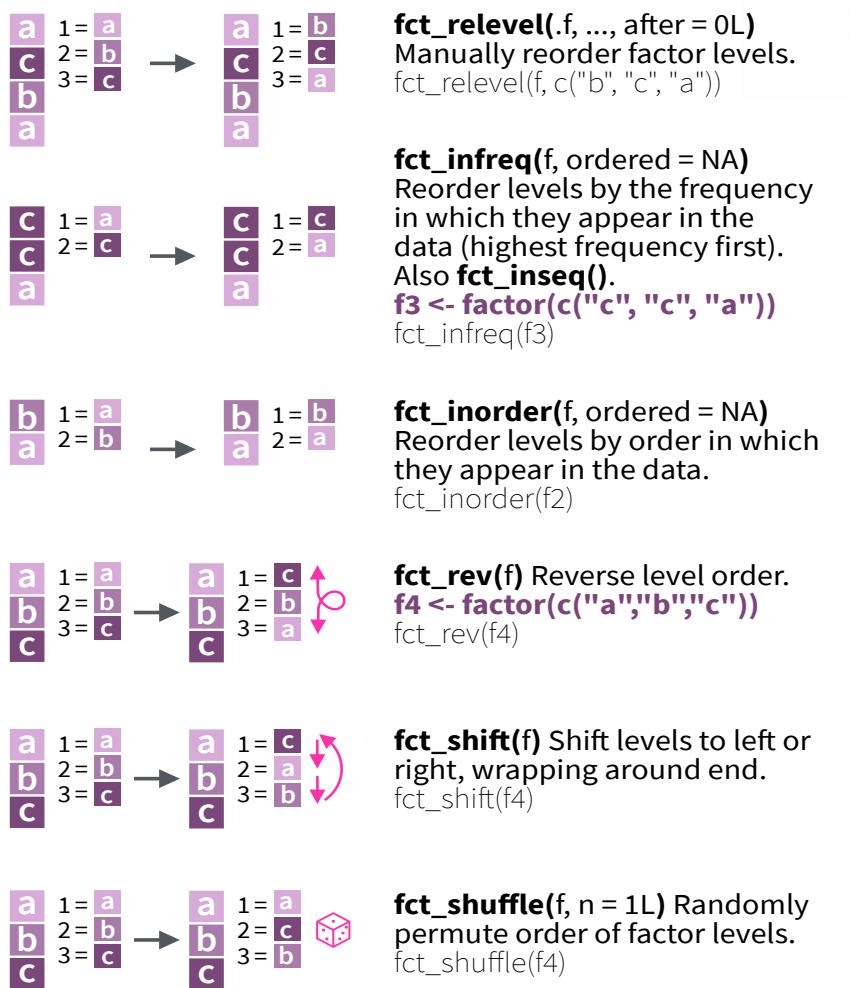
## Inspect Factors



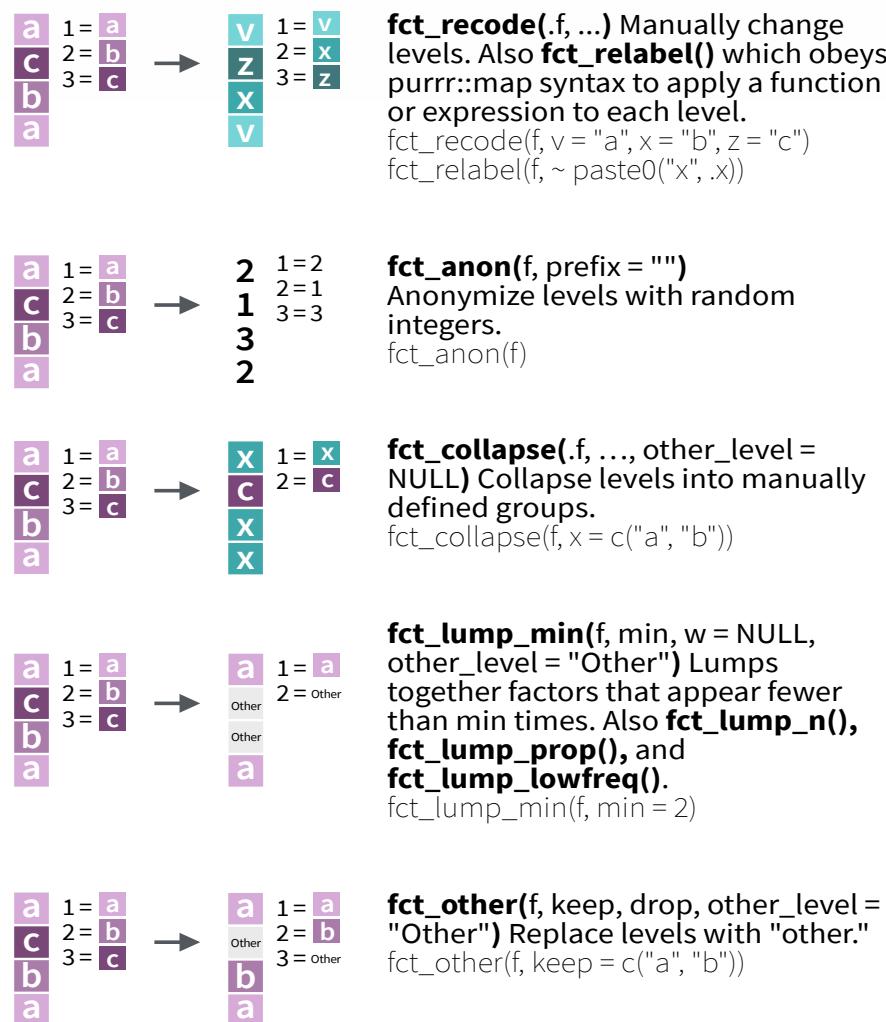
## Combine Factors



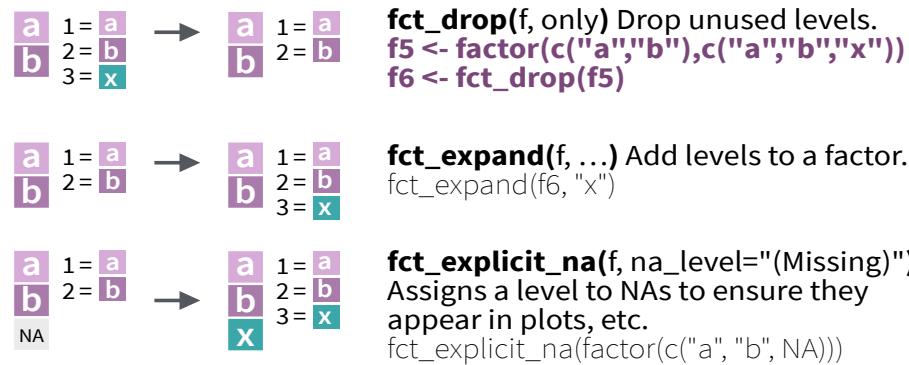
## Change the order of levels



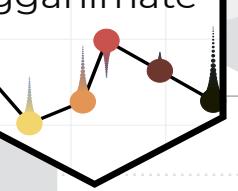
## Change the value of levels



## Add or drop levels



# Animate ggplots with gganimate :: CHEAT SHEET



## Core Concepts

gganimate builds on ggplot2's grammar of graphics to provide functions for animation. You add them to plots created with `ggplot()` the same way you add a geom.

### Main Function Groups

- `transition_*`(): What variable controls change and how?
- `view_*`(): Should the axes change with the data?
- `enter/exit_*`(): How does new data get added the plot? How does old data leave?
- `shadow_*`(): Should previous data be "remembered" and shown with current data?
- `ease_aes()`: How do you want to handle the pace of change between transition values?

**Note:** you only need a `transition_*`() or `view_*`() to make an animation. The other function groups enable you to add features or alter gganimate's default settings.

## Starting Plots

```
library(tidyverse)
library(gganimate)

a <- ggplot(diamonds,
 aes(carat, price)) +
 geom_point()

b <- ggplot(txhousing,
 aes(month, sales)) +
 geom_col()

c <- ggplot(economics,
 aes(date, psavert)) +
 geom_line()
```

## transition\_\*

`transition_states()`

```
a + transition_states(color, transition_length = 3, state_length = 1)
```

We're cycling between values of `color`, ...

... and spending **3** times as long going to the next cut as we do pausing there.

`transition_time()`

```
b + transition_time(year, range = c(2002L, 2006L))
```

We're cycling through each `year` of the data...

...from **2002** to **2006** (range is optional; default is the whole time frame). Unlike `transition_states()`, `transition_time()` treats the data as continuous and so the transition length is based on the actual values. Using **2002L** instead of **2002** because the underlying data is an integer.

`transition_reveal()`

```
c + transition_reveal(date)
```

We're adding each `date` of the data on top of 'old' data

`transition_length` and `filter_length` work the same as `transition/state_length()` in `transition_states()`...

`transition_filters()`

```
a + transition_filter(transition_length = 3,
 filter_length = 1,
 cut == "Ideal",
 Deep = depth >= 60)
```

... but now we're cycling between these two filtering conditions. **Names** are optional, but can be useful (see "Label variables" on next page).

### Other transitions

- `transition_manual()`: Similar to `transition_states()`, but without intermediate states.
- `transition_layers()`: Add layers (geoms) one at time.
- `transition_components()`: Transition elements independently from each other.
- `transition_events()`: Each element's duration can be controlled individually.

## Baseline Animation

```
anim_a <- a + transition_states(color, transition_length = 3, state_length = 1)
```

## view\_\*

`view_follow()`

```
anim_a +
 view_follow(fixed_x = TRUE,
 fixed_y = c(2500, NA))
```

x-axis shows **full range**, y shows **[2500, as much is needed for that frame]**. Default is for both axis to vary as needed.

`view_step()`

```
anim_a +
 view_step(pause_length = 2,
 step_length = 1,
 nstep = 7)
```

We're spending **twice** as long moving between views as staying at them...

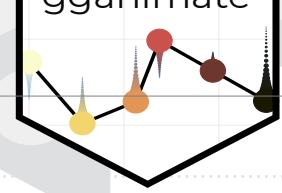
... and we're cycling between **seven** views. Seven is the number of steps in the transition, so the view is changing when the points are static, and visa versa. Views are determined by what data is in the current frame.

`view_zoom()`

`view_zoom()` works similarly to `view_step()`, except it changes the view by zooming and panning.

**Note:** both `view_step()` and `view_zoom()` have `view_*_manual()` versions for setting views directly instead of inferring it from frame data.

# Animate ggplots with gganimate :: CHEAT SHEET



## enter/\_exit\_\*

Every enter\_\*() function has a corresponding exit\_\*() function, and visa versa.

### enter/exit\_fade()

```
anim_a + enter_fade()
```

When new points need to be added, they will start transparent and become opaque.

### enter\_grow()/exit\_shrink()

```
anim_a + exit_shrink()
```

When extra points need to be removed, they will shrink in size before disappearing.

### enter/exit\_fly()

```
anim_a + enter_fly(x_loc = 0,
y_loc = 0)
```

When new points need to be added, they will fly in from **(0, 0)**.

### enter/exit\_drift()

```
anim_a + exit_drift(x_mod = 3, y_mod = -2)
```

When extra points need to be removed, They drift **3** units to the right and down **2** units before disappearing.

### enter/exit\_recolour() (or enter/exit\_recolor())

```
anim_a + enter_recolour(color = "red")
```

When new points need to be added, they start as **red** before transitioning to their correct color.

**Note:** enter/\_exit\_\*() functions can be combined so that you can have old data fade away and shrink to nothing by adding exit\_fade() and exit\_shrink() to the plot.

## shadow\_\*

### shadow\_wake()

```
anim_a + shadow_wake(wake_length = 0.05)
```

Points have a wake of points with the data from the last **5%** of frames.

### shadow\_trail()

```
anim_a + shadow_trail(distance = 0.05)
```

Animation will keep the points from **5%** of the frames, spaced as evenly as possible.

### shadow\_mark()

```
anim_a + shadow_mark(color = "red")
```

Animation will keep past states plotted in **red** (but not the intermediate frames).

## ease\_aes()

ease\_aes() allows you to set an easing function to control the rate of change between transition states. See ?ease\_aes for the full list.

Compare:

```
anim_a
```

```
anim_a + ease_aes("cubic-in") # Change easing of all aesthetics
```

```
anim_a + ease_aes(x = "elastic-in") # Only change `x` (others remain "linear")
```

## Saving animations

```
animation_to_save <- anim_a + exit_shrink()
anim_save("first_saved_animation.gif", animation = animation_to_save)
```

Since the animation argument uses your last rendered animation by default, this also works:

```
anim_a + exit_shrink()
anim_save("second_saved_animation.gif")
```

anim\_save() uses gifski to render the animation as a .gif file by default. You can use the renderer argument for other output types including video files (av\_renderer() or ffmpeg\_renderer()) or spritesheets (sprite\_renderer()):

```
requires you to have the av package installed
anim_save("third_saved_animation.mp4",
renderer = av_renderer())
```

## Label variables

gganimate's transition\_\*() functions create label variables you can pass to (sub)titles and other labels with the glue package. For example, transition\_states() has next\_state, which is the name of the state the animation is transitioning towards. Label variables are different between transitions, and details are included in the documentation of each.

```
anim_a + labs(subtitle = "Moving to {next_state}")
```

We're using the **next\_state** label variable to tell the viewer where we're going.

| Label variable                          | Description                                                                                                                             | Transitions              |
|-----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|--------------------------|
| transitioning                           | TRUE if the current frame is an transition frame, FALSE otherwise                                                                       | states, layers, filter   |
| previous_state/layer                    | Last shown state/layer                                                                                                                  | states, layers           |
| next_state/layer                        | State/layer that will been shown next                                                                                                   | states, layers           |
| closest_state/layer                     | State/layer that current frame is closest to (if between states/layers, either next or closest).                                        | states, layers           |
| previous/closest/next_filter/expression | Similar to their state/layer analogs.<br>*_filter variables return the name of the filter, *_expression variables return the condition. | filter                   |
| frame_time                              | Time of current frame                                                                                                                   | time, components, events |
| frame_along                             | Current frame's value for the dimension we're transitioning over                                                                        | reveal                   |
| nlayers                                 | Number of layers (total, not just currently shown)                                                                                      | layer                    |

# golem :: A Framework for Building Robust Shiny Apps

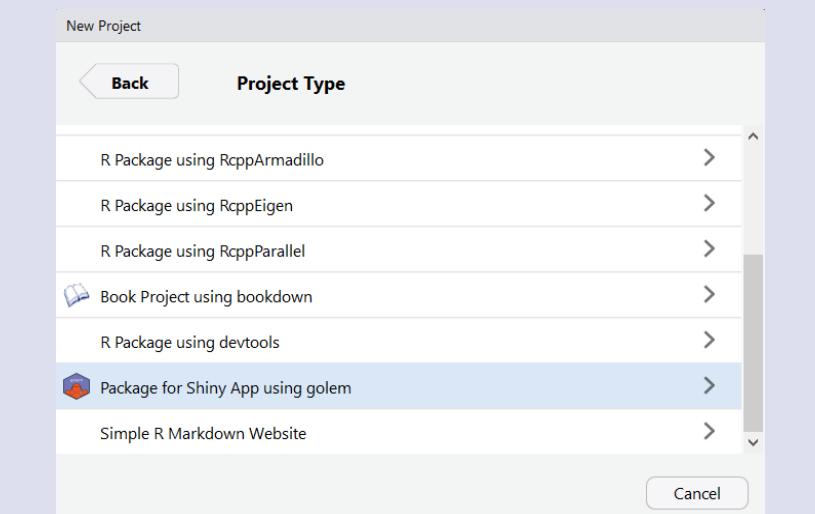
Create, maintain & deploy a packaged Shiny Application



## 1. Create a golem

With RStudio:

File ➔ New Project ➔ New Directory ➔ Shiny App using golem



Using the command line:

```
golem::create_golem(path = "~/appdemo")
Creates a golem at '~/appdemo'.
```

## 2. Set up your golem with dev/01\_start.R

```
golem::fill_desc(pkg_name = "appdemo", ...)
```

Fills the package DESCRIPTION with the author information, the application title & description, links...

```
golem::set_golem_options()
```

Sets {golem} global options.

```
golem::use_recommended_tests()
```

Creates a test template for your app.

```
golem::use_recommended_deps()
```

Adds {shiny}, {DT}, {attempt}, {glue}, {htmltools}, and {golem} as dependencies.

```
golem::use_favicon(path = "path/to/favicon.ico")
```

Changes the default favicon.

```
golem::use_utils_ui()
```

Creates 'R/golem\_utils\_ui.R', with UI-related helper functions.

```
golem::use_utils_server()
```

Creates 'R/golem\_utils\_server.R', with server-related helper functions.

## 3. Day-to-day dev with golem

### A. Look at your golem

- Launch your app with `dev/run_dev.R`:

```
options(golem.app.prod = FALSE)
Sets the prod or dev mode. (see ?golem::app_dev)

golem::detach_all_attached()
Detaches all loaded packages and cleans your environment.

golem::document_and_reload()
Documents and reloads your package.

appdemo::run_app()
Launches your application.
```

### B. Customise your golem with dev/02\_dev.R

- Edit `R/app_ui.R` & `R/app_server.R`

'R/app\_ui.R' & 'R/app\_server.R' hold the UI and server logic of your app. You can edit them directly, or add elements created with golem (e.g. modules).

- Add shiny modules

```
golem::add_module(name = "example")
Creates 'R/mod_example.R', with mod_example_ui and mod_example_server functions inside.
```

- Add external files

```
golem::add_js_file("script")
Creates 'inst/app/www/script.js'.
```

```
golem::add_js_handler("script")
Creates 'inst/app/www/script.js' with a skeleton for shiny custom handlers.
```

```
golem::add_css_file("custom")
Creates 'inst/app/www/custom.css'.
```

- Use golem built-in JavaScript functions

```
golem::activate_js()
Activates the built-in JavaScript functions. To be inserted in the UI.
```

```
golem::invoke_js("jsfunction", ns("ref_ui"))
Invokes from the server any JS function: built-in golem JS functions or custom ones created with add_js_handler()
```

## 4. Exhibit your golem

### Locally

```
remotes::install_local()
```

Installs your golem locally like any other package.

### To Rstudio products

```
golem::add_rstudioconnect_file()
```

Creates an app.R file, ready to be deployed to RStudio Connect.

```
golem::add_shinyappsio_file()
```

Creates an app.R file, ready to be deployed to shinyapps.io.

```
golem::add_shinyserver_file()
```

Creates an app.R file, ready to be deployed to Shiny Server.

### With Docker

```
golem::add_dockerfile()
```

Creates a Dockerfile that can launch your app.

```
golem::add_dockerfile_shinyproxy()
```

Creates a Dockerfile for ShinyProxy.

```
golem::add_dockerfile_heroku()
```

Creates a Dockerfile for Heroku.

## Tips and tricks

```
golem::print_dev("text")
```

Prints `text` in your console if `golem::app_dev()` is `TRUE`.

```
golem::make_dev(function)
```

Makes `function` depend on `golem::app_dev()` being `TRUE`.

```
golem::browser_button()
```

Creates a backdoor to your app (see `?golem::browser_button`).

- How to make a `run_dev` script for a specific module:

```
golem::detach_all_attached()
golem::document_and_reload()
```

```
ui <- mod_example_ui("my_module")
server <- function(input, output, session){
 callModule(mod_example_server, "my_module", session)
}
shinyApp(ui, server)
```

Keep in mind that a golem is a package. Everything you know about package development works with your packaged Shiny App created with `{golem}`! (documentation, tests, CI & CD, ...)



# GWAS Catalog access with gwasrapidd

## Introduction

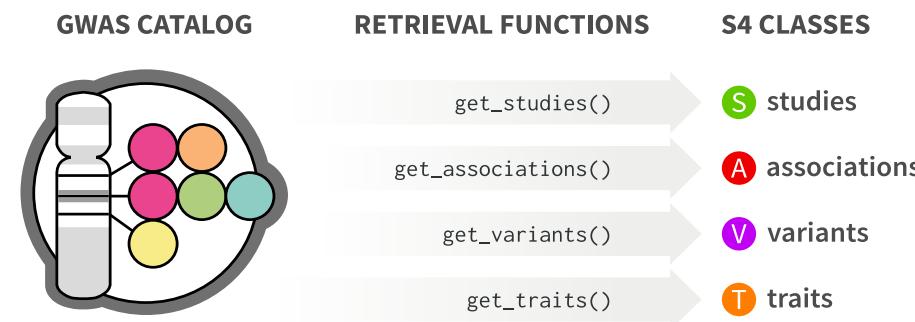
The **GWAS Catalog** is a service provided by the EMBL-EBI and NHGRI that offers a manually curated and freely available database of published genome-wide association studies (GWAS).

The GWAS Catalog data provided by the **RESTful API** is organized around four core entities:

- **studies**
- **associations**
- **variants**
- **traits**

## Get GWAS Catalog Entities

**gwasrapidd** facilitates the access to the Catalog via the RESTful API, allowing you to programmatically retrieve data directly into R. Each of the four entities is mapped to an S4 object of a class of the same name.



| Search by        | Example                                              | S A V T                                                                                                                                             |
|------------------|------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| study_id         | "GCST000858"                                         | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| association_id   | "24300113"                                           | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| variant_id       | "rs12752552"                                         | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| efo_id           | "EFO_0005543"                                        | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| pubmed_id        | "21626137"                                           | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| user_requested   | TRUE                                                 | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| full_pvalue_set  | FALSE                                                | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| efo_uri          | "http://www.ebi.ac.uk/efo/EF0_0004761"               | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| genomic_range    | list(chromosome = "22", start = 1L, end = 15473564L) | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| gene_name        | "BRCA1"                                              | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| efo_trait        | "lung adenocarcinoma"                                | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| reported_trait   | "Breast cancer"                                      | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |
| cytogenetic_band | "1p36.33"                                            | <span style="color: green;">●</span> <span style="color: red;">●</span> <span style="color: purple;">●</span> <span style="color: orange;">●</span> |

## S4 Representation of GWAS Catalog Entities

### S4 class studies

The **studies** object consists of eight slots, each a table (tibble). Each study is an observation (row) in the studies table — main table. All tables have the column `study_id` as primary key.

For details about the studies S4 class: `class?studies`.

| studies                                | genotyping_techs        | countries_of_recruitment    |
|----------------------------------------|-------------------------|-----------------------------|
| • <code>study_id</code>                | • <code>study_id</code> | • <code>study_id</code>     |
| • <code>reported_trait</code>          | • genotyping technology | • <code>ancestry_id</code>  |
| • <code>initial_sample_size</code>     | <b>platforms</b>        | • <code>country_name</code> |
| • <code>replication_sample_size</code> | • <code>study_id</code> | • <code>major_area</code>   |
| • <code>gxe</code>                     | • manufacturer          | • <code>region</code>       |
| • <code>gxg</code>                     | <b>ancestries</b>       | <b>countries_of_origin</b>  |
| • <code>snp_count</code>               | • <code>study_id</code> | • <code>study_id</code>     |
| • <code>qualifier</code>               | • ancestry_id           | • <code>ancestry_id</code>  |
| • <code>imputed</code>                 | • country_name          | • <code>country_name</code> |
| • <code>pooled</code>                  | • type                  | • <code>major_area</code>   |
| • <code>study_design_comment</code>    | • number_of_individuals | • <code>region</code>       |
| • <code>full_pvalue_set</code>         | <b>ancestral_groups</b> | <b>publications</b>         |
| • <code>user_requested</code>          | • <code>study_id</code> | • <code>study_id</code>     |
|                                        | • ancestry_id           | • <code>pubmed_id</code>    |
|                                        | • ancestral_group       | • publication_date          |
|                                        |                         | • publication               |
|                                        |                         | • title                     |
|                                        |                         | • author_fullname           |
|                                        |                         | • author_orcid              |

### S4 class associations

The **associations** object consists of six slots, each a table (tibble). Each association is an observation (row) in the associations table — main table. All tables have the column `association_id` as primary key.

For details about the associations S4 class: `class?associations`.

| associations                          | loci                               | genes                         |
|---------------------------------------|------------------------------------|-------------------------------|
| • <code>association_id</code>         | • <code>association_id</code>      | • <code>association_id</code> |
| • <code>pvalue</code>                 | • <code>locus_id</code>            | • <code>locus_id</code>       |
| • <code>pvalue_description</code>     | • <code>haplotype_snp_count</code> | • <code>gene_name</code>      |
| • <code>pvalue_mantissa</code>        | • <code>description</code>         | <b>ensembl_ids</b>            |
| • <code>pvalue_exponent</code>        | <b>risk_alleles</b>                | • <code>association_id</code> |
| • <code>multiple.snp.haplotype</code> | • <code>association_id</code>      | • <code>locus_id</code>       |
| • <code>snp_interaction</code>        | • <code>locus_id</code>            | • <code>variant_id</code>     |
| • <code>snp_type</code>               | • <code>variant_id</code>          | • <code>ensembl_id</code>     |
| • <code>standard_error</code>         | • <code>risk_allele</code>         | <b>entrez_ids</b>             |
| • <code>range</code>                  | • <code>risk_frequency</code>      | • <code>association_id</code> |
| • <code>or_per_copy_number</code>     | • <code>genome_wide</code>         | • <code>locus_id</code>       |
| • <code>beta_number</code>            | • <code>limited_list</code>        | • <code>gene_name</code>      |
| • <code>beta_unit</code>              |                                    | • <code>entrez_id</code>      |
| • <code>beta_direction</code>         |                                    |                               |
| • <code>beta_description</code>       |                                    |                               |
| • <code>last_mapping_date</code>      |                                    |                               |
| • <code>last_update_date</code>       |                                    |                               |

**A**

### S4 class variants

The **variants** object consists of four slots, each a table (tibble). Each variant is an observation (row) in the variants table — main table. All tables have the column `variant_id` as primary key.

For details about the variants S4 class: `class?variants`.

| variants                           | genomic_contexts                   | ensembl_ids               |
|------------------------------------|------------------------------------|---------------------------|
| • <code>variant_id</code>          | • <code>variant_id</code>          | • <code>variant_id</code> |
| • <code>merged</code>              | • <code>gene_name</code>           | • <code>gene_name</code>  |
| • <code>functional_class</code>    | • <code>chromosome_name</code>     | • <code>ensembl_id</code> |
| • <code>chromosome_name</code>     | • <code>chromosome_position</code> | • <code>entrez_ids</code> |
| • <code>chromosome_position</code> | • <code>distance</code>            | • <code>variant_id</code> |
| • <code>chromosome_region</code>   | • <code>is_closest_gene</code>     | • <code>gene_name</code>  |
| • <code>last_update_date</code>    | • <code>is_intergenic</code>       | • <code>entrez_id</code>  |
|                                    | • <code>is_upstream</code>         |                           |
|                                    | • <code>is_downstream</code>       |                           |
|                                    | • <code>source</code>              |                           |
|                                    | • <code>mapping_method</code>      |                           |

**V**

### S4 class traits

The **traits** object consists of one slot only, a table (tibble) of GWAS Catalog EFO traits. Each EFO trait is an observation (row) in the traits table — main table.

For details about the traits S4 class: `class?traits`.

| traits                |
|-----------------------|
| • <code>efo_id</code> |
| • <code>trait</code>  |
| • <code>uri</code>    |

**T**

## Manipulate Cases

Get a **studies** object **s** of two GWAS studies:

```
s <- get_studies(study_id = c('a', 'b'))
```

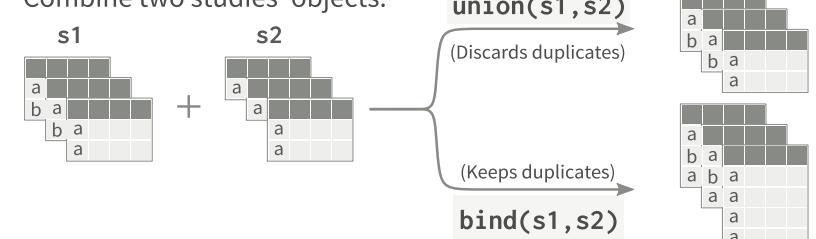
Subset object **s** by either identifier or position using '`[`:



```
s['a'] # Subset by identifier
```

```
s[1] # Subset by position
```

Combine two studies' objects:



# Dataset Operations

## DATA IMPORT / EXPORT

**h2o.uploadFile:** Upload a file into H2O from a client-side path, and parse it.

**h2o.downloadCSV:** Download a H2O dataset to a client-side CSV file.

**h2o.importFile:** Import a file into H2O from a server-side path, and parse it.

**h2o.exportFile:** Export an H2O Data Frame to a server-side file.

**h2o.parseRaw:** Parse a raw data file.

## NATIVE R TO H2O COERCION

**as.h2o:** Convert a R object to an H2O object

## H2O TO NATIVE R COERCION

**as.data.frame:** Check if an object is a data frame, and coerce it if possible.

## DATA GENERATION

**h2o.createFrame:** Creates a data frame in H2O with real-valued, categorical, integer, and binary columns specified by the user, with optional randomization.

**h2o.runif:** Produce a vector of random uniform numbers.

**h2o.interaction:** Create interaction terms between categorical features of an H2O Frame.

**h2o.target\_encode\_apply:** Target encoding map to an H2O Data Frame, which can improve performance of supervised learning models for high cardinality categorical columns.

## DATA SAMPLING / SPLITTING

**h2o.splitFrame:** Split an existing H2O dataset according to user-specified ratios.

## MISSING DATA HANDLING

**h2o.impute:** Impute a column of data using the mean, median, or mode.

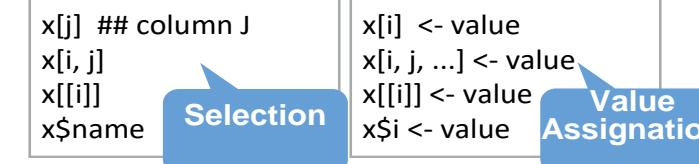
**h2o.insertMissingValues:** Replaces a user-specified fraction of entries in an H2O dataset with missing values.

**h2o.na.omit:** Remove Rows With NAs.

# General Operations

## SUBSCRIPTING

Subscripting example to pull (/push) pieces from (/to) a H2O Parsed Data object.



## SUBSETTING

**h2o.head, h2o.tail:** Object's Start or End.

## DATA ATTRIBUTES

**h2o.names:** Return column names for an H2O Frame. Also: **h2o.colnames**

**names<-:** Set the row or column names of a H2O Frame. Also: **colnames<-**

**h2o.dim:** Retrieve object dimensions.

**h2o.length:** Length of vector, list or factor.

**h2o.nrow:** Number of H2O Frame rows.

**h2o.ncol:** Number of H2O Frame columns.

**h2o.anyFactor:** Check if an H2O Frame object has any categorical data columns.

**is.factor, is.character, is.numeric:** Check Column's Data Type.

## DATA TYPE COERCION

Convert to:

**h2o.asfactor, as.factor:** Factor.

**h2o.as\_date, as.Date:** Date.

**h2o.ascharacter, as.character:** Character.

**h2o.asnumeric, as.numeric:** Numeric.

## BASIC DATA MANIPULATION

**c:** Combine Values into a Vector or List.

 **h2o.cbind; h2o.rbind:** Combine a sequence of H2O datasets by column (cbind) or rows (rbind).

 **h2o.merge:** Merges 2 H2OFrames.

 **h2o.arrange:** Sorts an H2OFrame by columns.

## ELEMENT INDEX SELECTION

**h2o.which:** True Condition's Row Numbers

## CONDITIONAL VALUE SELECTION

**h2o.ifelse:** Apply conditional statements to numeric vectors in an H2O Frame.

# Math Operations

## (math) vectorized function

### MATH

**h2o.abs:** Compute the absolute value of x.

**h2o.sqrt:** Principal Square Root of x,  $\sqrt{x}$ .

**h2o.ceiling:** Take a single numeric argument x and return a numeric vector containing the smallest integers not less than the corresponding elements of x.

**h2o.floor:** Take a single numeric argument x and return a numeric vector containing the largest integers not greater than the corresponding elements of x.

**h2o.trunc:** Take a single numeric argument x and return a numeric vector containing the integers formed by truncating the values in x toward 0.

**h2o.log:** Compute natural logarithms. See also: **h2o.log10, h2o.log2, h2o.log1p**

**h2o.exp:** Compute the exponential function

**h2o.cos, h2o.cosh, h2o.acos, h2o.sin, h2o.tan, h2o.tanh, Math:** ?groupGeneric

**sign:** Return a vector with the signs of the corresponding elements of x (the sign of a real number is 1, 0, or -1 if the number is positive, zero, or negative, respectively).

**&& (Vectorized AND), || (Vectorized OR), !x, %in%, Ops: +, -, \*, /, ^, %%, %/%, ==, !=, <, <=, >=, >, &, |, !**

## CUMULATIVE

**h2o.cummax:** Vector of the cumulative maxima of the elements of the argument.

**h2o.cummin:** Vector of the cumulative minima of the elements of the argument.

**h2o.cumprod:** Vector of the cumulative products of the elements of the argument.

**h2o.cumsum:** Vector of the cumulative sums of the elements of the argument.

## PRECISION

**h2o.round:** Round values to the specified number of decimal places. The default is 0.

**h2o.signif:** Round values to the specified number of significant digits.

# Group By Summaries

## (group by) summary function

**nrow:** Count the number of rows.

**max:** All input argument's Maximum.

**min:** All input argument's Minimum.

**sum:** All argument values Sum.

**mean:** (Trimmed) arithmetic mean.

**sd:** Calculate the standard deviation of a column of continuous real valued data.

**var:** Compute the variance of x.

# Generic Summaries

## NON-GROUP\_BY SUMMARIES

**h2o.median:** Calculate the median of x.

**h2o.range:** Input argument's Min/Max Vector

**h2o.cor:** Correlation Matrix of H2O Frames.

**h2o.quantile:** Obtain and display quantiles for an H2O Frame Column.

 **h2o.hist:** Compute a histogram over a numeric H2O Frame Column.

**h2o.prod:** Product of all arguments values.

**h2o.any:** Given a set of logical vectors, determine if at least one of the values is true.

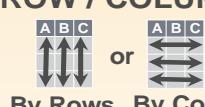
**h2o.all:** Given a set of logical vectors, determine if all of the values are true.

## NON-GROUP\_BY SUMMARIES: GENERIC

**h2o.summary:** Produce result summaries of the results of various model fitting functions.

# Aggregations

## ROW / COLUMN AGGREGATION

 **apply:** Apply a function over an H2O parsed data object (an array) margins.

## GROUP BY AGGREGATION

 **h2o.group\_by:** Apply an aggregate function to each group of an H2O dataset.

## TABULATION

 **h2o.table:** Use the cross-classifying factors to build a table of counts at each combination of factor levels.



# Data Modeling

## MODEL TRAINING: SUPERVISED LEARNING

**h2o.deeplearning:** Perform Deep Learning Neural Networks on an H2OFrame.

**h2o.gbm:** Build Gradient Boosted Regression Trees or Classification Trees.

**h2o.glm:** Fit a Generalized Linear Model, specified by a response variable, a set of predictors, and the error distribution.

**h2o.naiveBayes:** Compute Naive Bayes classification probabilities on an H2O Frame.

**h2o.randomForest:** Perform Random Forest Classification on an H2O Frame.

**h2o.xgboost:** Build an Extreme Gradient Boosted Model using the XGBoost backend.

**h2o.stackedEnsemble:** Build a stacked ensemble (aka. Super Learner) using the specified H2O base learning algorithms.

**h2o.automl:** Automates the Supervised Machine Learning Model Training Process: Automatically Trains and Cross-validates a set of Models, and trains a Stacked Ensemble.

## MODEL TRAINING: UNSUPERVISED LEARNING

**h2o.prcomp:** Perform Principal Components Analysis on the given H2O Frame.

**h2o.kmeans:** Perform k-means Clustering on the given H2O Frame.

**h2o.anomaly:** Detect anomalies in a H2O Frame using a H2O Deep Learning Model with Auto-Encoding.

**h2o.deepfeatures:** Extract the non-linear features from a H2O Frame using a H2O Deep Learning Model.

**h2o.glrm:** Builds a Generalized Low Rank Decomposition of an H2O Frame.

**h2o.svd:** Singular value decomposition of an H2O Frame using the power method.

**h2o.word2vec:** Trains a word2vec model on a String column of an H2O data frame.

## SURVIVAL MODELS: TIME-TO-EVENT

**h2o.coxph:** Trains a Cox Proportional Hazards Model (CoxPH) on an H2O Frame.

## GRID SEARCH

**h2o.grid:** Efficient method to build multiple models with different hyperparameters.

**h2o.getGrid:** Get a grid object from H2O distributed K/V store.

## MODEL SCORING

**h2o.predict:** Obtain predictions from various fitted H2O model objects.

**h2o.scoreHistory:** Get Model Score History.

## MODEL METRICS

**h2o.make\_metrics:** Given predicted values (target for regression, class-1 probabilities, or binomial or per-class probabilities for multinomial), compute a model metrics object.

## GENERAL MODEL HELPER

**h2o.performance:** Evaluate the predictive performance of a Supervised Learning Regression or Classification Model via various metrics. Set **xval = TRUE** for retrieving the training cross-validation metrics.

## REGRESSION MODEL HELPER

**h2o.mse:** Display the mean squared error calculated from "Predicted Responses" and "Actual (Reference) Responses". Set **xval = TRUE** for retrieving the cross-validation MSE.

## CLASSIFICATION MODEL HELPERS

**h2o.accuracy:** Get Model Accuracy metric.

**h2o.auc:** Retrieve the AUC (area under ROC curve). Set **xval = TRUE** for retrieving the cross-validation AUC.

**h2o.confusionMatrix:** Display prediction errors for classification data ("Predicted" vs "Reference : Real Values").

**h2o.hit\_ratio\_table:** Retrieve the Hit Ratios. Set **xval = TRUE** for retrieving the cross-validation Hit Ratio.

## CLUSTERING MODEL HELPER

**h2o.betweenss:** Get the between cluster Sum of Squares.

**h2o.centers:** Retrieve the Model Centers.

## PREDICTOR VARIABLE IMPORTANCE

**h2o.varimp:** Retrieve the variable importance

**h2o.varimp\_plot:** Plot Variable Importances.

# Data Munging

## GENERAL COLUMN MANIPULATION

**is.na:** Display missing elements.

## FACTOR LEVEL MANIPULATIONS

**h2o.levels:** Display a list of the unique values found in a categorical data column.

**h2o.relevel:** Reorders levels of an H2O factor, similarly to standard R's `relevel`.

**h2o.setLevels:** Set Levels of H2O Factor.

## NUMERIC COLUMN MANIPULATIONS

**h2o.cut:** Convert H2O Numeric Data to Factor by breaking it into Intervals.

## CHARACTER COLUMN MANIPULATIONS

**h2o.strsplit:** "String Split": Splits the given factor column on the input split.

**h2o.tolower:** Convert the characters of a character vector to lower case.

**h2o.toupper:** Convert the characters of a character vector to upper case.

**h2o.trim:** "Trim spaces": Remove leading and trailing white space.

**h2o.gsub:** Match a pattern & replace **all** instances (occurrences) of the matched pattern with the replacement string globally.

**h2o.sub:** Match a pattern & replace the **first** instance (occurrence) of the matched pattern with the replacement string.

## DATE MANIPULATIONS

**h2o.month:** Convert Milliseconds to Months in H2O Datasets (Scale: 0 to 11).

**h2o.year:** Convert Milliseconds to Years in H2O Datasets, indexed starting from 1900.

**h2o.day:** Convert Milliseconds to Day of Month in H2O Datasets (Scale: 1 to 31).

**h2o.hour:** Convert Milliseconds to Hour of Day in H2O Datasets (Scale: 0 to 23).

**h2o.dayOfWeek:** Convert Milliseconds to Day of Week in a H2OFrame (Scale: 0 to 6)

## MATRIX OPERATIONS

**%\*%:** Multiply two conformable matrices.

**t:** Returns the transpose of an H2OFrame.

# Cluster Operations

## H2O KEY VALUE STORE ACCESS

**h2o.assign:** Assign H2O hex.keys to R objects.

**h2o.getFrame:** Get H2O dataset Reference.

**h2o.getModel:** Get H2O model reference.

**h2o.ls:** Display a list of object keys in the running instance of H2O.

**h2o.rm:** Remove specified H2O Objects from the H2O server, but not from the R environment.

**h2o.removeAll:** Remove All H2O Objects from the H2O server, but not from the R environment.

## H2O MODEL IMPORT / EXPORT

**h2o.loadModel:** Load H2OModel from disk.

**h2o.saveModel:** Save H2OModel object to disk.

**h2o.download\_pojo:** Download the Scoring POJO (Plain Old Java Object) of an H2O Model.

**h2o.download\_mojo:** Download the model in MOJO format.

## H2O CLUSTER CONNECTION

**h2o.init:** Connect to a running H2O instance using all CPUs on the host.

**h2o.shutdown:** Shut down the specified H2O instance. All data on the server will be lost!

## H2O CLUSTER INFORMATION

**h2o.clusterInfo:** Display the name, version, uptime, total nodes, total memory, total cores and health of a cluster running H2O.

**h2o.clusterStatus:** Retrieve information on the status of the cluster running H2O.

## H2O LOGGING

**h2o.clearLog:** Clear all H2O R command and error response logs from the local disk.

**h2o.downloadAllLogs:** Download all H2O log files to the local disk.

**h2o.logAndEcho:** Write a message to the H2O Java log file and echo it back.

**h2o.openLog:** Open existing logs of H2O R POST commands and error responses on disk.

**h2o.getLogPath:** Get the file path for the H2O R command and error response logs.

**h2o.startLogging:** Begin logging H2O R POST commands and error responses.

**h2o.stopLogging:** Stop logging H2O R POST commands and error responses.

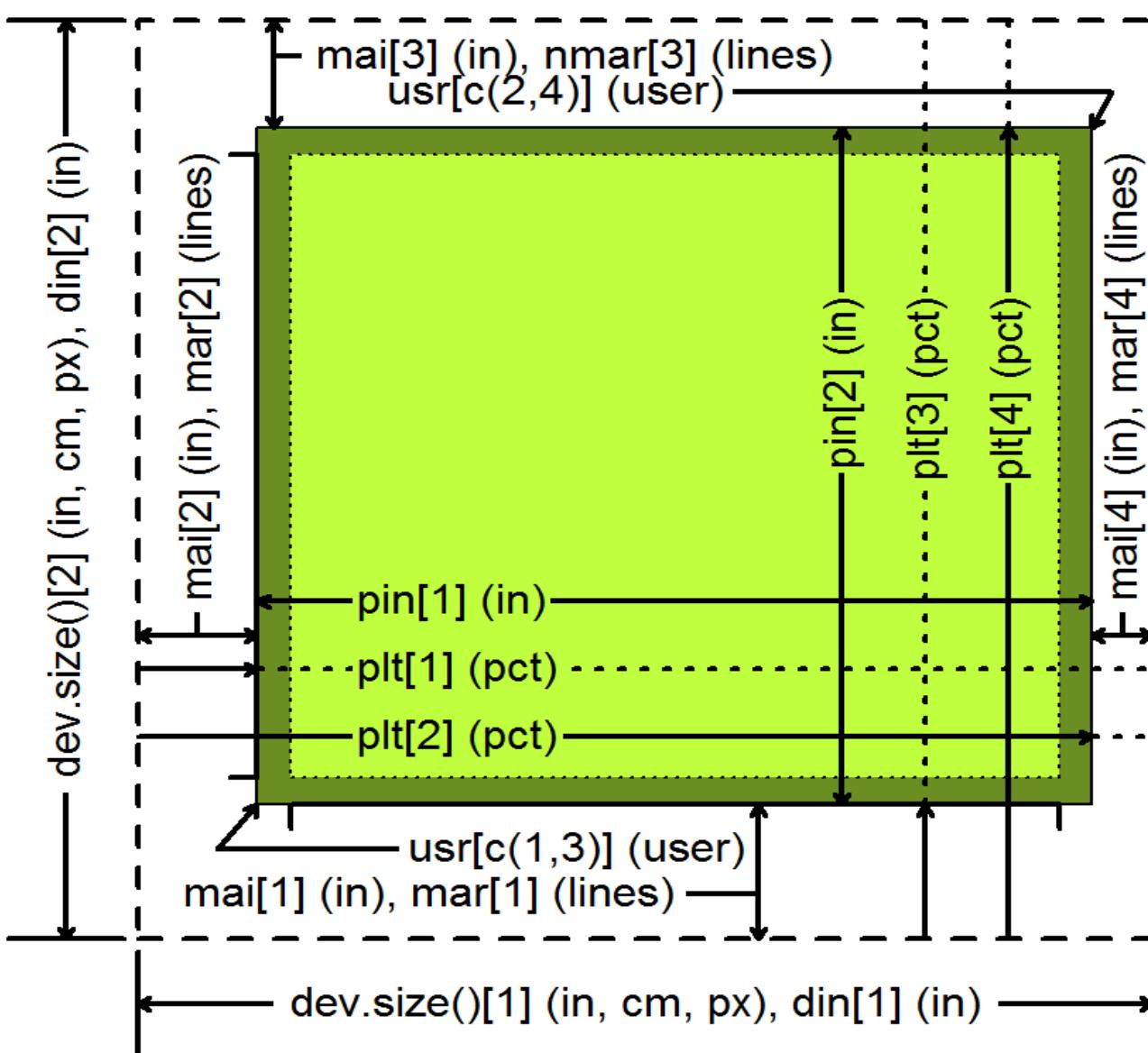


# How Big is Your Graph?

## An R Cheat Sheet

### Introduction

All functions that open a device for graphics will have **height** and **width** arguments to control the size of the graph and a **pointsize** argument to control the relative font size. In **knitr**, you control the size of the graph with the chunk options, **fig.width** and **fig.height**. This sheet will help you with calculating the size of the graph and various parts of the graph within R.



### Your graphics device

`dev.size()` (width, height)  
`par("din")` (r.o.) (width, height) in inches

Both the **dev.size** function and the **din** argument of **par** will tell you the size of the graphics device. The **dev.size** function will report the size in

1. inches (**units="in"**), the default
2. centimeters (**units="cm"**)
3. pixels (**units="px"**)

Like several other **par** arguments, **din** is read only (r.o.) meaning that you can ask its current value (`par("din")`) but you cannot change it (`par(din=c(5,7))` will fail).

### Your plot margins

`par("mai")` (bottom, left, top, right) in inches  
`par("mar")` (bottom, left, top, right) in lines

Margins provide you space for your axes, axis labels, and titles.

A "line" is the amount of vertical space needed for a line of text.

If your graph has no axes or titles, you can remove the margins (and maximize the plotting region) with

```
par(mar=rep(0,4))
```

### Your plotting region

`par("pin")` (width, height) in inches  
`par("plt")` (left, right, bottom, top) in pct

The **pin** argument **par** gives you the size of the plotting region (the size of the device minus the size of the margins) in inches.

The **plt** argument gives you the percentage of the device from the left/bottom edge up to the left edge of the plotting region, the right edge, the bottom edge, and the top edge. The first and third values are equivalent to the percentage of space devoted to the left and bottom margins. Subtract the second and fourth values from 1 to get the percentage of space devoted to the right and top margins.

### Your x-y coordinates

`par("usr")` (xmin, ymin, xmax, ymax)

Your x-y coordinates are the values you use when plotting your data. This normally is not the same as the values you specified with the **xlim** and **ylim** arguments in **plot**. By default, R adds an extra 4% to the plotting range (see the dark green region on the figure) so that points right up on the edges of your plot do not get partially clipped. You can override this by setting **xaxs="i"** and/or the **yaxs="i"** in **par**.

Run `par("usr")` to find the minimum X value, the maximum X value, the minimum Y value, and the maximum Y value. If you assign new values to **usr**, you will update the x-y coordinates to the new values.

### Getting a square graph

`par("pty")`

You can produce a square graph manually by setting the width and height to the same value and setting the margins so that the sum of the top and bottom margins equal the sum of the left and right margins. But a much easier way is to specify **pty="s"**, which adjusts the margins so that the size of the plotting region is always square, even if you resize the graphics window.

### Converting units

For many applications, you need to be able to translate user coordinates to pixels or inches. There are some cryptic shortcuts, but the simplest way is to get the range in user coordinates and measure the proportion of the graphics device devoted to the plotting region.

```
user.range <- par("usr")[c(2,4)] -
 par("usr")[c(1,3)]
```

```
region.pct <- par("plt")[c(2,4)] -
 par("plt")[c(1,3)]
```

```
region.px <-
 dev.size(units="px") * region.pct
```

```
px.per.xy <- region.px / user.range
```

To convert a horizontal or distance from the x-coordinate value to pixels, multiply by **px.per.xy[1]**. To convert a vertical distance, multiply by **region.px.per.xy[2]**. To convert a diagonal distance, you need to invoke Pythagoras.

```
a.px <- x.dist*px.per.xy[1]
b.px <- y.dist*px.per.xy[2]
c.px <- sqrt(a.px^2+b.px^2)
```

To rotate a string to match the slope of a line segment, you need to convert the distances to pixels, calculate the arctangent, and convert from radians to degrees.

```
segments(x0, y0, x1, y1)
delta.x <- (x1 - x0) * px.per.xy[1]
delta.y <- (y1 - y0) * px.per.xy[2]
angle.radians <- atan2(delta.y, delta.x)
angle.degrees <- angle.radians * 180 / pi
text(x1, y1, "TEXT", srt=angle.degrees)
```

## Panels

`par("fig")` (width, height) in pct  
`par("fin")` (width, height) in inches

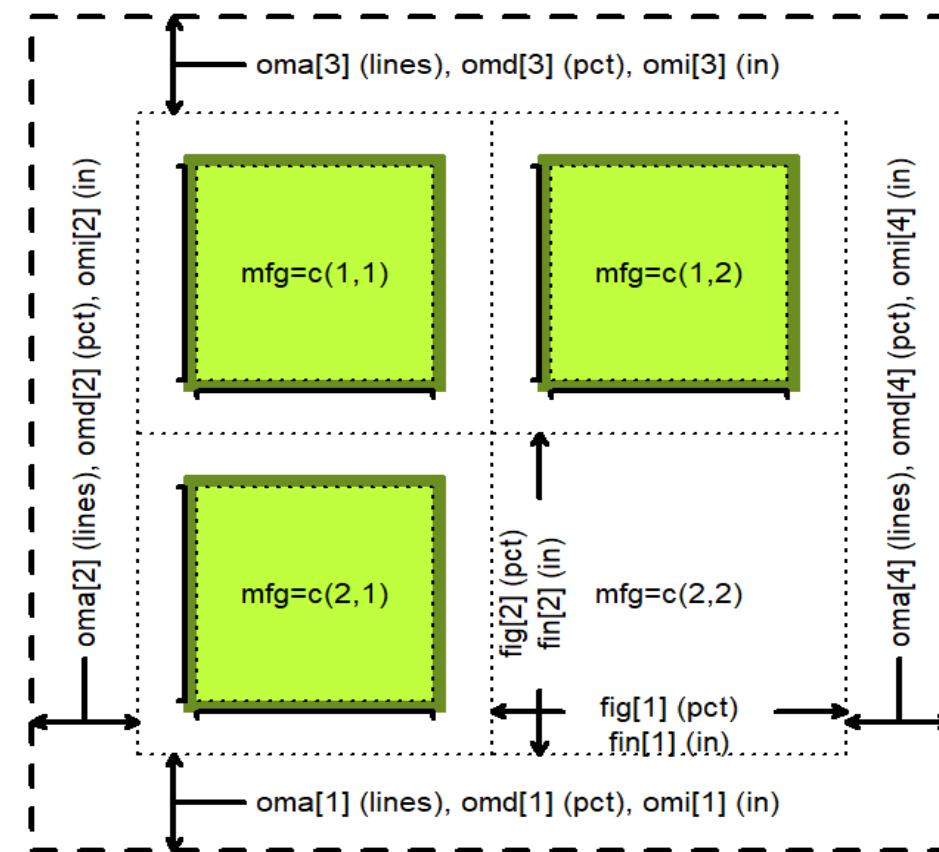
If you display multiple plots within a single graphics window (e.g., with the `mfrow` or `mfcol` arguments of `par` or with the `layout` function), then the `fig` and `fin` arguments will tell you the size of the current subplot window in percent or inches, respectively.

`par("oma")` (bottom, left, top, right) in lines  
`par("omd")` (bottom, left, top, right) in pct  
`par("omi")` (bottom, left, top, right) in inches

Each subplot will have margins specified by `mai` or `mar`, but no outer margin around the entire set of plots, unless you specify them using `oma`, `omd`, or `omi`. You can place text in the outer margins using the `mtext` function with the argument `outer=TRUE`.

`par("mfg")` (r, c) or (r, c, maxr, maxc)

The `mfg` argument of `par` will allow you to jump to a subplot in a particular row and column. If you query with `par("mfg")`, you will get the current row and column followed by the maximum row and column.



## Character and string sizes

### `strheight()`

The `strheight` functions will tell you the height of a specified string in inches (`units="inches"`), x-y user coordinates (`units="user"`) or as a percentage of the graphics device (`units="figure"`).

For a single line of text, `strheight` will give you the height of the letter "M". If you have a string with one or more linebreaks ("\n"), the `strheight` function will measure the height of the letter "M" plus the height of one or more additional lines. The height of a line is dependent on the line spacing, set by the `lheight` argument of `par`. The default line height (`lheight=1`), corresponding to single spaced lines, produces a line height roughly 1.5 times the height of "M".

### `strwidth()`

The `strwidth` function will produce different widths to individual characters, representing the proportional spacing used by most fonts (a "W" using much more space than an "i"). For the width of a string, the `strwidth` function will sum up the lengths of the individual characters in the string.

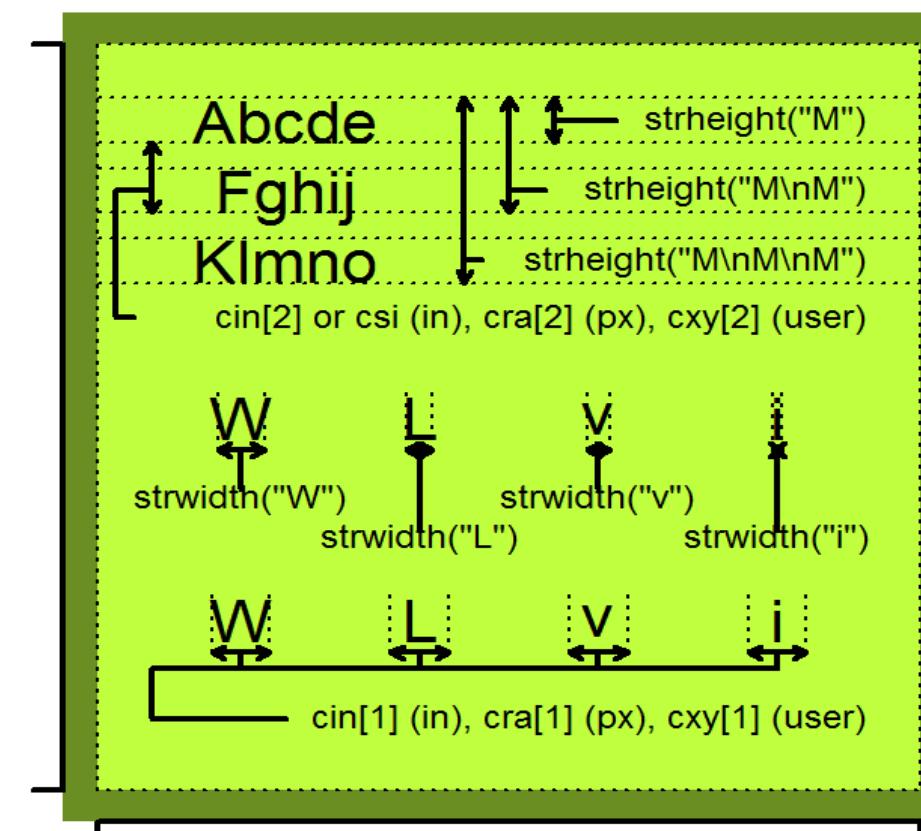
`par("cin")` (r.o.) (width, height) in inches  
`par("csi")` (r.o.) height in inches  
`par("cra")` (r.o.) (width, height) in pixels  
`par("cxy")` (r.o.) (width, height) in xy coordinates

The single value returned by the `csi` argument of `par` gives you the height of a line of text in inches. The second of the two values returned by `cin`, `cra`, and `cxy` gives you the height of a line, in inches, pixels, or xy (user) coordinates.

The first of the two values returned by the `cin`, `cra`, and `cxy` arguments to `par` gives you the approximate width of a single character, in inches, pixels, or xy (user) coordinates. The width, very slightly smaller than the actual width of the letter "W", is a rough estimate at best and ignores the variable width of individual letters.

These values are useful, however, in providing fast ratios of the relative sizes of the differing units of measure

```
px.per.in <- par("cra") / par("cin")
px.per.xy <- par("cra") / par("cxy")
xy.per.in <- par("cxy") / par("cin")
```



## If your fonts are too big or too small

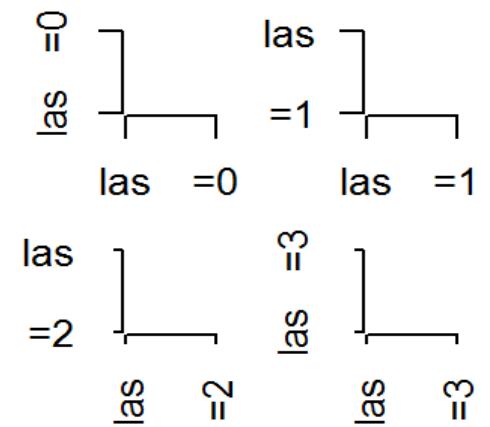
Fixing this takes a bit of trial and error.

- Specify a larger/smaller value for the `pointsize` argument when you open your graphics device.
- Try opening your graphics device with different values for `height` and `width`. Fonts that look too big might be better proportioned in a larger graphics window.
- Use the `cex` argument to increase or decrease the relative size of your fonts.

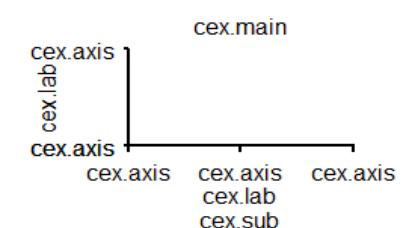
## If your axes don't fit

There are several possible solutions.

- You can assign wider margins using the `mar` or `mai` argument in `par`.
- You can change the orientation of the axis labels with `las`. Choose among
  - `las=0` both axis labels parallel
  - `las=1` both axis labels horizontal
  - `las=2` both axis labels perpendicular
  - `las=3` both axis labels vertical.



- change the relative size of the font
  - `cex.axis` for the tick mark labels.
  - `cex.lab` for `xlab` and `ylab`.
  - `cex.main` for the main title
  - `cex.sub` for the subtitle.



# Mission

Missing Data is nearly everywhere. Also in time series, especially in sensor recordings missing data is common.

*imputeTS helps you with your missing data problems.*

# Features

The package provides easy to use functions in these areas:

## 1. Imputation Functions

Several algorithms for replacing NAs with reasonable values (imputation).



## 2. Missing Data Visualizations

Plots for analysis of the distribution of NAs, patterns and imputation performance.



## 3. Stats and Datasets

Functions for printing missing data stats and benchmarking datasets.



# Scope

imputeTS specializes on univariate time series that are:

- numeric
- equally-spaced



# Visualizations

There are multiple plots provided for analysing the missing data before and after imputation. All plotting functions start with `ggplot_na_`plotname

| Function                            | Description                            |
|-------------------------------------|----------------------------------------|
| <code>ggplot_na_distribution</code> | Getting a first overview of NAs        |
| <code>ggplot_na_intervals</code>    | Insights about NAs in specific periods |
| <code>ggplot_na_gapsize</code>      | Insights about occurring NA gapsizes   |
| <code>ggplot_na_imputations</code>  | Evaluating imputation quality          |

# Imputation

The package offers multiple missing data replacement (imputation) functions, which are really easy to use.

```
na_interpolation(x, option = "spline")
```

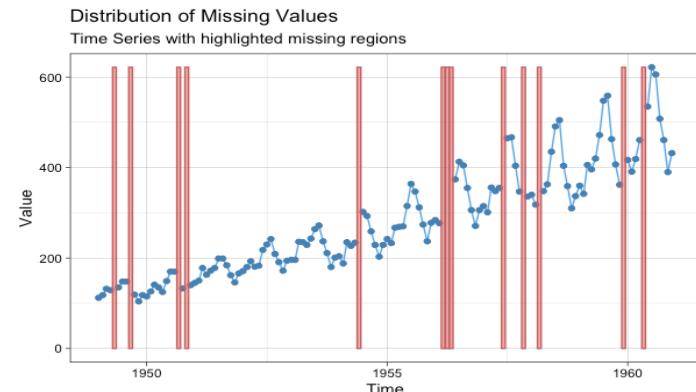
Imputation Function    Your input time series    Additional Parameters

## List of available Algorithms

| Function                      | Description                               |
|-------------------------------|-------------------------------------------|
| <code>na_interpolation</code> | Imputation by Interpolation               |
| <code>na_kalman</code>        | Imputation by Kalman Smoothing            |
| <code>na_locf</code>          | Last Observation Carried Forward          |
| <code>na_ma</code>            | Imputation by Moving Average              |
| <code>na_mean</code>          | Imputation by Mean Value                  |
| <code>na_random</code>        | Imputation by Random Sample               |
| <code>na_remove</code>        | Remove Missing Values                     |
| <code>na_replace</code>       | Replace Missing Values by a Defined Value |
| <code>na_seadec</code>        | Seasonally Decomposed Imputation          |
| <code>na_seasplit</code>      | Seasonally Splitted Imputation            |

# Missing Data Overview Plots

The ‘distribution’, ‘intervals’ and ‘gapsize’ plots can be used on new datasets to gain insights about missing data patterns and distribution.

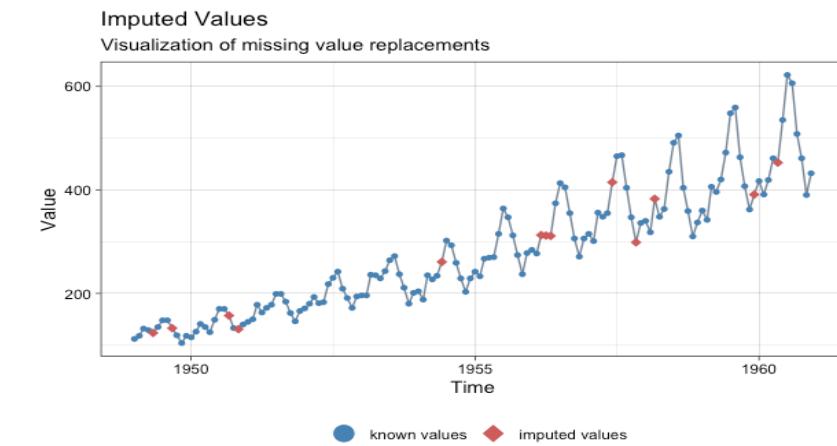


```
ggplot_na_distribution(tsAirgap)
```

example data with NAs

# Imputation Analysis Plots

Imputation results can be visualized with the ‘imputations’ plot. Here first `na_kalman` is performed and then the results are plotted.



```
imp <- na_kalman(tsAirgap)
ggplot_na_imputations(tsAirgap, imp)
```

can be put in pipe workflows  
a 36 step forecast is created and plotted

# Workflows

The functions also work well in tidy style pipe workflows. Here an example of first using imputation and later forecasting and plotting.

```
library("forecast")
tsAirgap %>% na_interpolation() %>%
 ets() %>% forecast(h=36) %>%
 autoplot()
```

can be put in pipe workflows  
a 36 step forecast is created and plotted

# Datasets

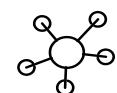
The package includes three datasets for imputation experiments.

| Function               | Description                                                                      |
|------------------------|----------------------------------------------------------------------------------|
| <code>tsAirgap</code>  | Monthly totals of international airline passengers.<br>144 Observations / 13 NAs |
| <code>tsNH4</code>     | NH4 concentration in a wastewater system.<br>3552 observations / 883 NAs         |
| <code>tsHeating</code> | A heating systems supply temperature.<br>606837 observations / 57391 NAs         |

## CITATION

You can cite `imputeTS` the following:

Moritz, Steffen and Bartz-Beielstein, Thomas.  
"imputeTS: Time Series Missing Value Imputation in R."  
R Journal 9.1 (2017). doi: 10.32614/RJ-2017-009.



# Audit sampling with jfa::: CHEAT SHEET



## Basics

jfa is an R package that facilitates statistical planning, selection, and evaluation of audit samples.

The package provides five functions that allow users to easily apply Bayesian or classical probability theory in the standard audit sampling workflow.

## Installation

Installing the package can be done via:

```
install.packages('jfa')
```

Loading the package can be done via:

```
library(jfa)
```

## Example

The blue code blocks next to the function descriptions provide a working example of the intended workflow.

The data for this example can be loaded via:

```
data('BuildIt')
```



## Construct a prior probability distribution (optional)

```
jfa:::auditPrior()
```

This function creates a prior distribution for the misstatement in the population based on audit evidence specified via the `method` argument. The prior distribution can be used as input for the `prior` argument in other functions to perform Bayesian inference.

```
auditPrior(method = 'default',
 likelihood = 'poisson', ...)
```

## Calculate the minimum sample size

```
jfa:::planning()
```

Given a performance materiality or a minimum precision, this function calculates the minimum sample size to achieve these objectives based on the binomial, Poisson, or hypergeometric `likelihood`. `prior` can be specified to perform Bayesian planning.

- `expected`: A fraction or an integer specifying the expected errors in the sample.

```
planning(materiality = 0.05,
 expected = 0.01,
 likelihood = 'poisson',
 conf.level = 0.95,
 prior = FALSE, ...)
```

## Select the required items from the population

```
jfa:::selection()
```

This function takes a data frame and performs sampling according to one of three popular `method`'s: random sampling, cell sampling, or fixed interval sampling. Sampling is done in combination with one of two sampling `units`: items (rows) or monetary units.

```
selection(data = BuildIt,
 size = 93,
 units = 'items',
 method = 'interval', ...)
```

## Evaluate the misstatement in the population

```
jfa:::evaluation()
```

This function takes a data sample (using `data`, `values`, and `values.audit`) or summary statistics from a sample (using `x` and `n`) and performs statistical evaluation on the misstatement in the population according to the specified `method`. `A prior` can be specified to perform Bayesian evaluation.

- `prior`: An object returned by `auditPrior()` that specifies the prior distribution.

```
evaluation(materiality = 0.05,
 method = 'poisson',
 alternative = 'less',
 conf.level = 0.95,
 x = 0,
 n = 93,
 prior = FALSE, ...)
```

## Create a report of the statistical results

```
jfa:::report()
```

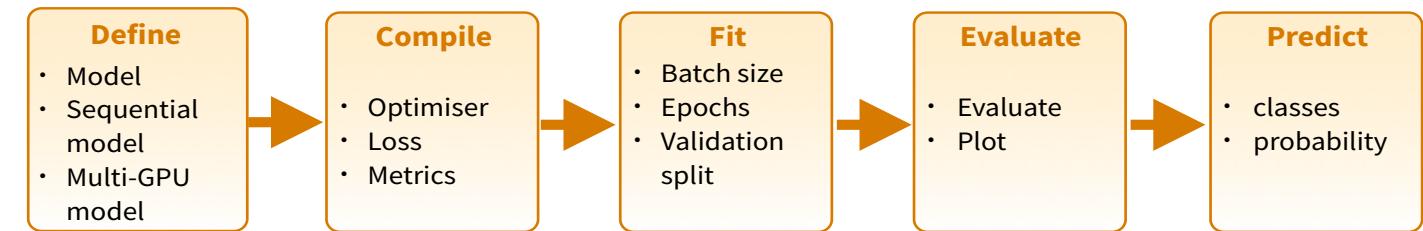
This function takes an object of class `jfaEvaluation` as returned by `evaluation()` and automatically generates a report containing the statistical results and their interpretation.

```
report(object = evaluationResult,
 file = 'report.html', ...)
```

# Intro

Keras is a high-level neural networks API developed with a focus on enabling fast experimentation. It supports multiple backends, including TensorFlow, CNTK and Theano.

TensorFlow is a lower level mathematical library for building deep neural network architectures. The keras R package makes it easy to use Keras and TensorFlow in R.



<https://keras.rstudio.com>

<https://www.manning.com/books/deep-learning-with-r>

The “Hello, World!”  
of deep learning

# Working with keras models

## DEFINE A MODEL

`keras_model()` Keras Model

`keras_model_sequential()` Keras Model composed of a linear stack of layers

`multi_gpu_model()` Replicates a model on different GPUs

## COMPILE A MODEL

`compile(object, optimizer, loss, metrics = NULL)`

Configure a Keras model for training

## FIT A MODEL

`fit(object, x = NULL, y = NULL, batch_size = NULL, epochs = 10, verbose = 1, callbacks = NULL, ...)`  
Train a Keras model for a fixed number of epochs (iterations)

`fit_generator()` Fits the model on data yielded batch-by-batch by a generator

`train_on_batch(); test_on_batch()` Single gradient update or model evaluation over one batch of samples

## EVALUATE A MODEL

`evaluate(object, x = NULL, y = NULL, batch_size = NULL)` Evaluate a Keras model

`evaluate_generator()` Evaluates the model on a data generator

## PREDICT

`predict()` Generate predictions from a Keras model

`predict_proba() and predict_classes()`  
Generates probability or class probability predictions for the input samples

`predict_on_batch()` Returns predictions for a single batch of samples

`predict_generator()` Generates predictions for the input samples from a data generator

## OTHER MODEL OPERATIONS

`summary()` Print a summary of a Keras model

`export_savedmodel()` Export a saved model

`get_layer()` Retrieves a layer based on either its name (unique) or index

`pop_layer()` Remove the last layer in a model

`save_model_hdf5(); load_model_hdf5()` Save/Load models using HDF5 files

`serialize_model(); unserialize_model()`  
Serialize a model to an R object

`clone_model()` Clone a model instance

`freeze_weights(); unfreeze_weights()`  
Freeze and unfreeze weights

## CORE LAYERS



`layer_input()` Input layer



`layer_dense()` Add a densely-connected NN layer to an output



`layer_activation()` Apply an activation function to an output



`layer_dropout()` Applies Dropout to the input



`layer_reshape()` Reshapes an output to a certain shape



`layer_permute()` Permute the dimensions of an input according to a given pattern



`layer_repeat_vector()` Repeats the input n times



`layer_lambda(object, f)` Wraps arbitrary expression as a layer



`layer_activity_regularization()`  
Layer that applies an update to the cost function based input activity



`layer_masking()` Masks a sequence by using a mask value to skip timesteps



`layer_flatten()` Flattens an input

## INSTALLATION

The keras R package uses the Python keras library. You can install all the prerequisites directly from R.

[https://keras.rstudio.com/reference/install\\_keras.html](https://keras.rstudio.com/reference/install_keras.html)

`library(keras)`  
`install_keras()`

See `?install_keras`  
for GPU instructions

This installs the required libraries in an Anaconda environment or virtual environment 'r-tensorflow'.

## TRAINING AN IMAGE RECOGNIZER ON MNIST DATA

### # input layer: use MNIST images

```
mnist <- dataset_mnist()
x_train <- mnist$train$x; y_train <- mnist$train$y
x_test <- mnist$test$x; y_test <- mnist$test$y
```



### # reshape and rescale

```
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
x_train <- x_train / 255; x_test <- x_test / 255
```

```
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
```

### # defining the model and layers

```
model <- keras_model_sequential()
model %>%
 layer_dense(units = 256, activation = 'relu',
 input_shape = c(784)) %>%
 layer_dropout(rate = 0.4) %>%
 layer_dense(units = 128, activation = 'relu') %>%
 layer_dense(units = 10, activation = 'softmax')
```



### # compile (define loss and optimizer)

```
model %>% compile(
 loss = 'categorical_crossentropy',
 optimizer = optimizer_rmsprop(),
 metrics = c('accuracy'))
```

### # train (fit)

```
model %>% fit(
 x_train, y_train,
 epochs = 30, batch_size = 128,
 validation_split = 0.2
)
model %>% evaluate(x_test, y_test)
model %>% predict_classes(x_test)
```

# Pre-trained models

Keras applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning.

**application\_xception()**  
**xception\_preprocess\_input()**  
Xception v1 model

**application\_inception\_v3()**  
**inception\_v3\_preprocess\_input()**  
Inception v3 model, with weights pre-trained on ImageNet

**application\_inception\_resnet\_v2()**  
**inception\_resnet\_v2\_preprocess\_input()**  
Inception-ResNet v2 model, with weights trained on ImageNet

**application\_vgg16(); application\_vgg19()**  
VGG16 and VGG19 models

**application\_resnet50()** ResNet50 model

**application\_mobilenet()**  
**mobilenet\_preprocess\_input()**  
**mobilenet\_decode\_predictions()**  
**mobilenet\_load\_model\_hdf5()**  
MobileNet model architecture

## IMAGENET

[ImageNet](#) is a large database of images with labels, extensively used for deep learning

**imagenet\_preprocess\_input()**  
**imagenet\_decode\_predictions()**  
Preprocesses a tensor encoding a batch of images for ImageNet, and decodes predictions

## Callbacks

A callback is a set of functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training.

**callback\_early\_stopping()** Stop training when a monitored quantity has stopped improving  
**callback\_learning\_rate\_scheduler()** Learning rate scheduler  
**callback\_tensorboard()** TensorBoard basic visualizations

## CONVOLUTIONAL LAYERS



**layer\_conv\_1d()** 1D, e.g. temporal convolution



**layer\_conv\_2d\_transpose()** Transposed 2D (deconvolution)



**layer\_conv\_2d()** 2D, e.g. spatial convolution over images



**layer\_conv\_3d\_transpose()** Transposed 3D (deconvolution)  
**layer\_conv\_3d()** 3D, e.g. spatial convolution over volumes



**layer\_conv\_lstm\_2d()** Convolutional LSTM



**layer\_separable\_conv\_2d()** Depthwise separable 2D



**layer\_upsampling\_1d()**  
**layer\_upsampling\_2d()**  
**layer\_upsampling\_3d()**  
Upsampling layer



**layer\_zero\_padding\_1d()**  
**layer\_zero\_padding\_2d()**  
**layer\_zero\_padding\_3d()**  
Zero-padding layer



**layer\_cropping\_1d()**  
**layer\_cropping\_2d()**  
**layer\_cropping\_3d()**  
Cropping layer

## POOLING LAYERS



**layer\_max\_pooling\_1d()**  
**layer\_max\_pooling\_2d()**  
**layer\_max\_pooling\_3d()**  
Maximum pooling for 1D to 3D



**layer\_average\_pooling\_1d()**  
**layer\_average\_pooling\_2d()**  
**layer\_average\_pooling\_3d()**  
Average pooling for 1D to 3D



**layer\_global\_max\_pooling\_1d()**  
**layer\_global\_max\_pooling\_2d()**  
**layer\_global\_max\_pooling\_3d()**  
Global maximum pooling



**layer\_global\_average\_pooling\_1d()**  
**layer\_global\_average\_pooling\_2d()**  
**layer\_global\_average\_pooling\_3d()**  
Global average pooling

## ACTIVATION LAYERS



**layer\_activation()** object, activation  
Apply an activation function to an output



**layer\_activation\_leaky\_relu()**  
Leaky version of a rectified linear unit



**layer\_activation\_parametric\_relu()**  
Parametric rectified linear unit



**layer\_activation\_thresholded\_relu()**  
Thresholded rectified linear unit



**layer\_activation\_elu()**  
Exponential linear unit

## DROPOUT LAYERS



**layer\_dropout()**  
Applies dropout to the input



**layer\_spatial\_dropout\_1d()**  
**layer\_spatial\_dropout\_2d()**  
**layer\_spatial\_dropout\_3d()**  
Spatial 1D to 3D version of dropout

## RECURRENT LAYERS



**layer\_simple\_rnn()**  
Fully-connected RNN where the output is to be fed back to input



**layer\_gru()**  
Gated recurrent unit - Cho et al



**layer\_cudnn\_gru()**  
Fast GRU implementation backed by CuDNN



**layer\_lstm()**  
Long-Short Term Memory unit - Hochreiter 1997



**layer\_cudnn\_lstm()**  
Fast LSTM implementation backed by CuDNN

## LOCALLY CONNECTED LAYERS

**layer\_locally\_connected\_1d()**  
**layer\_locally\_connected\_2d()**

Similar to convolution, but weights are not shared, i.e. different filters for each patch

## SEQUENCE PREPROCESSING

**pad\_sequences()**

Pads each sequence to the same length (length of the longest sequence)

**skipgrams()**

Generates skipgram word pairs

**make\_sampling\_table()**

Generates word rank-based probabilistic sampling table

## TEXT PREPROCESSING

**text\_tokenizer()** Text tokenization utility

**fit\_text\_tokenizer()** Update tokenizer internal vocabulary

**save\_text\_tokenizer(); load\_text\_tokenizer()**

Save a text tokenizer to an external file

**texts\_to\_sequences();**

**texts\_to\_sequences\_generator()**

Transforms each text in texts to sequence of integers

**texts\_to\_matrix(); sequences\_to\_matrix()**

Convert a list of sequences into a matrix

**text\_one\_hot()** One-hot encode text to word indices

**text\_hashing\_trick()**

Converts a text to a sequence of indexes in a fixed-size hashing space

**text\_to\_word\_sequence()**

Convert text to a sequence of words (or tokens)

## IMAGE PREPROCESSING

**image\_load()** Loads an image into PIL format.

**flow\_images\_from\_data()**

**flow\_images\_from\_directory()**

Generates batches of augmented/normalized data from images and labels, or a directory

**image\_data\_generator()** Generate minibatches of image data with real-time data augmentation.

**fit\_image\_data\_generator()** Fit image data generator internal statistics to some sample data

**generator\_next()** Retrieve the next item

**image\_to\_array(); image\_array\_resize()**

**image\_array\_save()** 3D array representation



R Studio

## Basics

Labelled data is a common data structure in other statistical environment such as Stata, SAS or SPSS.

It consists of a set of additional attributes for numeric and character vectors (including columns of a data frame).

There are 3 types of attributes:

1. **Variable labels** (a short description of a variable)
2. **Value labels** (labels associated to specific values)
3. **Missing values:**
  - User-defined missing values (SPSS style)
  - Tagged NA (Stata and SAS style)

## Variable labels

### MANIPULATING A VECTOR

- var\_label(x) or var\_label(df\$v1)**  
Get the variable label associated to a vector x
- var\_label(x) <- "variable description"**  
Add/modify a variable label to x
- var\_label(x) <- NULL**  
Remove the variable label associated to x

### MANIPULATING A DATA.FRAME

- var\_label(df)**  
List all variable labels associated with columns of df
- var\_label(df) <- list(v1 = "variable 1", v2 = "variable 2")**  
Update variable labels of some columns of df
- df %>% set\_variable\_labels(v1 = "variable 1", v2 = "variable 2", v3 = NULL)**  
Update variable labels using dplyr syntax
- df %>% look\_for()**  
Return a data frame with all variable names and labels
- df %>% look\_for("s")**  
Search variables containing "s" in their name or label
- df %>% look\_for(details = TRUE)**  
Return additional details on each variable

## Value labels

When value labels are attached to a numeric or character vector, the vector's class becomes **haven\_labelled**. A major difference with a factor is that values of the vector are not changed and it is not mandatory to attach a label to each value.

### MANIPULATING A VECTOR

- val\_label(x, value) or val\_label(df\$v1, value)**  
Get the label attached to a specific value of a vector
- val\_label(x, value) <- "label"**  
Set/Update the label attached to a specific value
- val\_label(x, value) <- NULL**  
Remove the label attached to a specific value
- val\_labels(x)**  
Get all value labels attached to a vector
- val\_labels(x) <- c(no = 0, yes = 1, maybe = 9)**  
Set/Update all value labels attached to a vector
- val\_labels(x) <- NULL**  
Remove all value labels attached to a vector
- labelled(c("F", "F", "M"), c(Female = "F", Male = "M"))**  
Create a labelled vector
- sort\_val\_labels(x, according\_to = "values")**  
Sort value labels according to values (or labels)
- drop\_unused\_value\_labels(x)**  
Remove value labels not observed in the data

### MANIPULATING A DATA.FRAME

- df %>% set\_value\_labels(v1 = c(Yes = 1, No = 2), v2 = c(Male = "M", Female = "F"))**  
Define value labels of several variables
- df %>% add\_value\_labels(v1 = c(Unknown = 9))**  
Add specific value labels to a variable (other already defined value labels remains unchanged)
- df %>% remove\_value\_labels(v1 = 9)**  
Remove specific value labels to a variable
- df %>% set\_value\_labels(v1 = NULL)**  
Remove all value labels attached to a variable
- df %>% drop\_unused\_value\_labels()**  
Remove value labels not observed in the data

## Missing values

### USER-DEFINED MISSING VALUES (SPSS STYLE)

Used to indicate that some values should be considered as missing. However, they will not be treated as NA as long as they are not converted to proper NA.

When missing values are attached to a numeric or character vector, the vector's class becomes **haven\_labelled\_spss**.

When importing a SPSS file, use the option `user_na = TRUE` to keep defined missing values (otherwise, they will be converted to NA).

### na\_values(x)

Get individual missing values attached to a vector

### na\_values(x) <- c(8, 9, 10)

df %>% set\_na\_values(v1 = c(8, 9, 10))  
Set/Update individual missing values (NULL to remove)

### na\_range(x)

Get a range of missing values attached to a vector

### na\_range(x) <- c(8, 10)

df %>% set\_na\_range(v1 = c(8, 10))  
Set/Update a range of missing values (NULL to remove)

### user\_na\_to\_na(x) or df %>% user\_na\_to\_na()

Convert user-defined missing values to NA

### is\_na(x)

TRUE if NA or if a user-defined missing value

### TAGGED NAs (STATA & SAS STYLE)

"Tagged" missing values work exactly like regular R missing values except that they store one additional byte of information: a tag, which is usually a letter ("a" to "z").

x <- c(1:5, tagged\_na("a"), tagged\_na("z"), NA)

**tagged\_na("a")** generates a NA with a tag

### is.na(x)

Tagged NAs work identically to regular NAs

### is\_tagged\_na(x)

Test if it is a tagged NA

### na\_tag(x)

Display the tags associated to tagged NAs

### format\_tagged\_na(x)

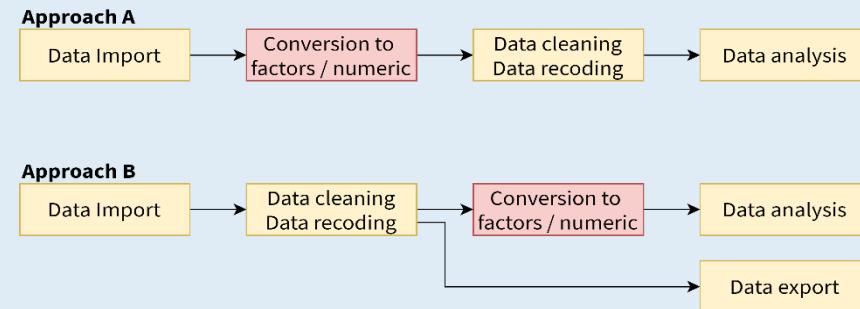
Convert x to a character vector showing the tagged NAs

`haven_labelled` and `haven_labelled_spss` classes introduced in **haven** package allow to add metadata (variable labels, value labels and SPSS-style missing values) to vectors / data frame columns and to properly import these metadata from SAS, Stata or SPSS.

Functions and methods provided by **labelled** package are designed for easy manipulation of such labelled data.

It should be noted that **value labels** doesn't imply that your vectors should be considered as categorical or continuous. Therefore, value labels are not intended to be used for data analysis. For example, before performing modeling or plotting, you should convert vectors with value labels into factors or into classic numeric/character vectors.

Two main approaches could be considered:



In **approach A**, labelled vectors are converted into factors or into numeric/character vectors just after data import, using `unlabelled()`, `to_factor()` or `unclass()`. Then, data cleaning, data recoding and analysis are performed using classic R vector types.

In **approach B**, labelled vectors are kept for data cleaning and recoding, allowing to preserve original coding, in particular if data should be reexported after that step. Functions provided by **labelled** will be useful for managing value labels.

However, as in approach A, labelled vectors will have to be converted into classic factors or numeric vectors before data analysis as this is the way categorical and continuous variables should be coded for analysis.

## OF LABELLED VECTORS

If all value labels and user-defined missing values are removed from a labelled vector, the `haven_labelled` class will be removed and the vector will be transformed into a basic numeric or character vector. **Values of the vector will remain unchanged.**

### `remove_val_labels(x)`

Remove value labels attached to a vector.

### `remove_user_na(x)`

Remove user-defined (`na_values` and `na_range`) from a vector

### `unclass(x)`

Remove the `haven_labelled` class. Therefore, the vector will be considered as a classical numeric or character vector. Value labels and user-defined missing values will still be visible as attributes attached to the vector.

When converting a labelled vector into a factor or a character vector of the value labels, be aware that **original values of the vector will be converted**.

### `to_character(x)`

Convert into a character vector replacing values by their corresponding value label

### `to_factor(x)`

Convert into a character vector replacing values by their corresponding value label

### `to_factor(x, levels = "prefixed")`

Value labels will be prefixed with their original value

### `to_factor(x, strict = TRUE)`

Convert into a factor only if all observed values have a value label

### `df %>% to_factor()`

Convert all labelled vectors and only labelled vectors into factors

### `df %>% to_factor(labelled_only = FALSE)`

Convert all columns (including non labelled vectors) into factors

### `unlabelled(x)`

### `df %>%>% unlabelled()`

Labelled vectors will be converted into factors only if all observed values have a value label. Otherwise, they will be unclassed. Similar to `df %>% to_factor(labelled_only = T, strict = T, unclass = T)`

### `to_factor(x, drop_unused_labels = TRUE)`

### `df %>% unlabelled(drop_unused_labels = TRUE)`

Unused value labels will be dropped before conversion into factors

## INTO LABELLED VECTORS

### `to_labelled(f)`

Convert a factor into a numeric labelled vector. Note that `to_labelled(to_factor(x))` and `x` will not be identical (original coding will be lost).

### `to_labelled(df)`

If df was imported with the **foreign** package or if it is a data set created with **memisc** package, meta data (variable labels, value labels and user-defined missing values) will be converted into **labelled** format.

If any value label or user-defined missing value is added to a numeric or a character vector, it will be automatically converted into a labelled vector.

**Values of the vector will remain unchanged.**

## Miscellaneous

### `nolabel_to_na(x) or df %>% nolabel_to_na()`

For labelled vectors, values without a value label will be converted into NA

### `val_labels_to_na(x) or df %>% val_labels_to_na()`

For labelled vectors, values with a value label will be converted into NA

### `df2 %>% copy_labels_from(df1)`

Copy variable labels, values labels and user-defined missing values from df1 to df2 based on shared columns names. Useful when attributes are lost after some data manipulation.

### `recode(x, `2` = 1, `3` = 2)`

Apply dplyr::recode() to a labelled vector. Attached value labels will remain unchanged.

### `recode(x, `2` = 1, .combine_value_labels = TRUE)`

This option will combine value labels of original values merged together to produce new value labels. It is recommended to check that the result is appropriate.

### `update_labelled(x) or df %>% update_labelled()`

If x/df was imported/created using an older version of **haven** or **labelled**, you may encounter some unexpected results. `update_labelled()` will update all labelled vectors to be consistent with the current implementation.

# Leaflet Cheat Sheet



for   
an open-source JavaScript library for mobile-friendly interactive maps

## Quick Start

### Installation

Use `install.packages("leaflet")` to install the package or directly from Github `devtools::install_github("rstudio/leaflet")`.

### First Map

```
m <- leaflet() %>%
 addTiles() %>%
 addMarkers lng = 174.768, lat = -36.852, popup = "The birthplace of R"
 # add a single point layer
```



## Map Widget

### Initialization

```
m <- leaflet(options = leafletOptions(...))
center Initial geographic center of the map
zoom Initial map zoom level
minZoom Minimum zoom level of the map
maxZoom Maximum zoom level of the map
```

### Map Methods

```
m %>% setView(lng, lat, zoom, options = list())
 Set the view of the map (center and zoom level)
m %>% fitBounds(lng1, lat1, lng2, lat2)
 Fit the view into the rectangle [lng1, lat1] - [lng2, lat2]
m %>% clearBounds()
 Clear the bound, automatically determine from the map elements
```

### Data Object

Both `leaflet()` and the `map` layers have an optional data parameter that is designed to receive spatial data with the following formats:

#### Base R

*The arguments of all layers take normal R objects:*  
`df <- data.frame(lat = ..., lng = ...)`

#### sp package

*Useful functions:*

`SpatialPoints, SpatialLines, SpatialPolygons, ...`

#### maps package

*Build a map of states with colors:*

`mapStates <- map("state", fill = TRUE, plot = FALSE)`

`leaflet(mapStates) %>% addTiles() %>%`

`addPolygons(fillColor = topo.colors(10, alpha = NULL), stroke = FALSE)`

## Markers

Use markers to call out points, express locations with latitude/longitude coordinates, appear as icons or as circles.  
Data come from vectors or assigned data frame, or `sp` package objects.

### Icon Markers

*Regular Icons: default and simple*

`addMarkers(lng, lat, popup, label)` *add basic icon markers*

`makeIcon/icons` (`iconUrl`, `iconWidth`, `iconHeight`, `iconAnchorX`, `iconAnchorY`, `shadowUrl`, `shadowWidth`, `shadowHeight`, ...) *customize marker icons*

`iconList()` *create a list of icons*

*Awesome Icons: customizable with colors and icons*

`addAwesomeMarkers`, `makeAwesomeIcon`, `awesomelcons`, `awesomelconList`

*Marker Clusters: option of addMarkers()*

`clusterOptions = markerClusterOptions()`

`freezeAtZoom` *Freeze the cluster at assigned zoom level*

### Circle Markers

`addCircleMarkers(color, radius, stroke, opacity, ...)`

*Customize their color, radius, stroke, opacity*

## Popups and Labels

`addPopups(lng, lat, ...content..., options)` *Add standalone popups*

`options = popupOptions(closeButton=FALSE)`

`addMarkers(..., popup, ...)` *Show popups with markers or shapes*

`addMarkers(..., label, labelOptions...)` *Show labels with markers or shapes*

`labelOptions = labelOptions(noHide, textOnly, textSize, direction, style)`

`addLabelOnlyMarkers()` *Add labels without markers*

## Lines and Shapes

### Polygons and Polylines

`addPolygons(color, weight=1, smoothFactor=0.5, opacity=1.0, fillOpacity=0.5, fillColor= ~colorQuantile("YlOrRd", ALAND)(ALAND), highlightOptions, ...)`

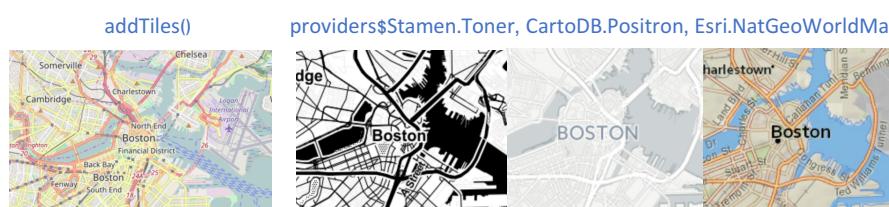
`highlightOptions(color, weight=2, bringToFront=TRUE)` *highlight shapes*

Use `rmapshaper::ms_simplify` to simplify complex shapes

*Circles* `addCircles(lng, lat, weight=1, radius, ...)`

*Rectangles* `addRectangles(lng1, lat1, lng2, lat2, fillColor="transparent", ...)`

## Basemaps



Default Tiles

Use `addTiles()` to add a custom map tile URL template, use `addWMSTiles()` to add WMS (Web Map Service) tiles

## GeoJSON and TopoJSON

There are two options to use the GeoJSON/TopoJSON data.

\* To read into `sp` objects with the `geojsonio` or `rgdal` package:  
`geojsonio::geojson_read(..., what="sp") rgdal::readOGR(..., "OGRGeoJSON")`

\* Or to use the `addGeoJSON()` and `addTopoJSON()` functions:  
`addTopoJSON/addGeoJSON(... weight, color, fill, opacity, fillOpacity... )`

Styles can also be tuned separately with a `style: {}` object.

Other packages including `RJSONIO` and `jsonlite` can help fast parse or generate the data needed.

## Shiny Integration

To integrate a Leaflet map into an app:

\* In the UI, call `leafletOutput("name")`

\* On the server side, assign a `renderLeaflet(...)` call to the output

\* Inside the `renderLeaflet` expression, return a Leaflet map object

### Modification

To modify an existing map or add incremental changes to the map, you can use `leafletProxy()`. This should be performed in an observer on the server side.

Other useful functions to edit your map:

`fitBounds(o, 0, 11, 11)` *similar to setView*

*fit the view to within these bounds*

`addCircles(1:10, 1:10, layerId = LETTERS[1:10])`

*create circles with layerIds of "A", "B", "C"...*

`removeShape(c("B", "F"))` *remove some of the circles*

`clearShapes()` *clear all circles (and other shapes)*

### Inputs/Events

#### Object Events

Object event names generally use this pattern:

`input$MAPID_OBJCATEGORY_EVENTNAME`

Triger an event changes the value of the Shiny input at this variable.

Valid values for `OBJCATEGORY` are `marker`, `shape`, `geojson` and `topojson`.

Valid values for `EVENTNAME` are `click`, `mouseover` and `mouseout`.

All of these events are set to either `NULL` if the event has never happened, or a `list()` that includes:

\* `lat` The latitude of the object, if available; otherwise, the mouse cursor

\* `lng` The longitude of the object, if available; otherwise, the mouse cursor

\* `id` The `layerId`, if any

GeoJSON events also include additional properties:

\* `featureId` The feature ID, if any

\* `properties` The feature properties

#### Map Events

`input$MAPID_click` *when the map background or basemap is clicked*

*value -- a list with lat and lng*

`input$MAPID_bounds` *provide the lat/lng bounds of the visible map area*

*value -- a list with north, east, south and west*

`input$MAPID_zoom` *an integer indicates the zoom level*

# Date-times



**2017-11-28 12:00:00**

**2017-11-28 12:00:00**

A **date-time** is a point on the timeline, stored as the number of seconds since 1970-01-01 00:00:00 UTC

```
dt <- as_datetime(1511870400)
"2017-11-28 12:00:00 UTC"
```

## PARSE DATE-TIMES

(Convert strings or numbers to date-times)

- Identify the order of the year (**y**), month (**m**), day (**d**), hour (**h**), minute (**m**) and second (**s**) elements in your data.
- Use the function below whose name replicates the order. Each accepts a tz argument to set the time zone, e.g. ymd(x, tz = "UTC").

**2017-11-28T14:02:00**

**ymd\_hms()**, **ymd\_hm()**, **ymd\_h()**.  
ymd\_hms("2017-11-28T14:02:00")

**2017-22-12 10:00:00**

**ydm\_hms()**, **ydm\_hm()**, **ydm\_h()**.  
ydm\_hms("2017-22-12 10:00:00")

**11/28/2017 1:02:03**

**mdy\_hms()**, **mdy\_hm()**, **mdy\_h()**.  
mdy\_hms("11/28/2017 1:02:03")

**1 Jan 2017 23:59:59**

**dmy\_hms()**, **dmy\_hm()**, **dmy\_h()**.  
dmy\_hms("1 Jan 2017 23:59:59")

**20170131**

**ymd()**, **ydm()**. ymd(20170131)

**July 4th, 2000**

**mdy()**, **myd()**. mdy("July 4th, 2000")

**4th of July '99**

**dmy()**, **dym()**. dmy("4th of July '99")

**2001: Q3**

**yq()** Q for quarter. yq("2001: Q3")

**07-2020**

**my()**, **ym()**. my("07-2020")

**2:01**

**hms::hms()** Also **lubridate::hms()**, **hm()** and **ms()**, which return periods.\* **hms::hms(sec = 0, min = 1, hours = 2, roll = FALSE)**

**2017.5**

**date\_decimal(decimal, tz = "UTC")**  
date\_decimal(2017.5)



**now(tzone = "")** Current time in tz (defaults to system tz). now()

**today(tzone = "")** Current date in a tz (defaults to system tz). today()

**fast.strptime()** Faster strftime.  
fast.strptime('9/1/01', '%y/%m/%d')

**parse\_date\_time()** Easier strftime.  
parse\_date\_time("9/1/01", "ymd")

**2017-11-28**

A **date** is a day stored as the number of days since 1970-01-01

```
d <- as_date(17498)
"2017-11-28"
```

## GET AND SET COMPONENTS

Use an accessor function to get a component. Assign into an accessor function to change a component in place.

**2018-01-31 11:59:59**

**2018-01-31 11:59:59**

**2018-01-31 11:59:59**

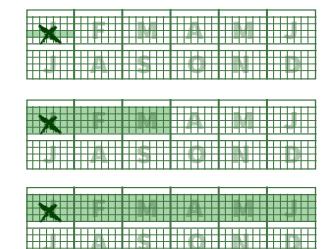
**2018-01-31 11:59:59**

**2018-01-31 11:59:59**

**2018-01-31 11:59:59**

**2018-01-31 11:59:59**

**2018-01-31 11:59:59 UTC**



**12:00:00**

An **hms** is a **time** stored as the number of seconds since 00:00:00

```
t <- hms::as_hms(85)
00:01:25
```

```
d ## "2017-11-28"
day(d) ## 28
day(d) <- 1
d ## "2017-11-01"
```

**date(x)** Date component. date(dt)

**year(x)** Year. year(dt)

**isoyear(x)** The ISO 8601 year.

**epiyear(x)** Epidemiological year.

**month(x, label, abbr)** Month.

month(dt)

**day(x)** Day of month. day(dt)

**wday(x, label, abbr)** Day of week.

**qday(x)** Day of quarter.

**hour(x)** Hour. hour(dt)

**minute(x)** Minutes. minute(dt)

**second(x)** Seconds. second(dt)

**tz(x)** Time zone. tz(dt)

**week(x)** Week of the year. week(dt)

**isoweek()** ISO 8601 week.

**epiweek()** Epidemiological week.

**quarter(x)** Quarter. quarter(dt)

**semester(x, with\_year = FALSE)**

Semester. semester(dt)

**am(x)** Is it in the am? am(dt)

**pm(x)** Is it in the pm? pm(dt)

**dst(x)** Is it daylight savings? dst(d)

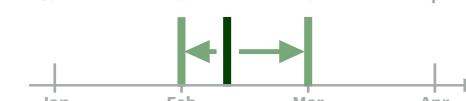
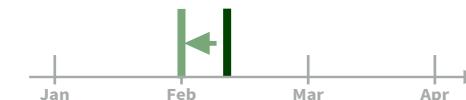
**leap\_year(x)** Is it a leap year?

leap\_year(d)

**update(object, ..., simple = FALSE)**

update(dt, mday = 2, hour = 1)

# Round Date-times



Valid units are second, minute, hour, day, week, month, bimonth, quarter, season, halfyear and year.

**rollback(dates, roll\_to\_first = FALSE, preserve\_hms = TRUE)** Roll back to last day of previous month. Also **rollforward()**. rollback(dt)

# Stamp Date-times

**stamp()** Derive a template from an example string and return a new function that will apply the template to date-times. Also **stamp\_date()** and **stamp\_time()**.

- Derive a template, create a function  
sf <- stamp("Created Sunday, Jan 17, 1999 3:34")
- Apply the template to dates  
sf(ymd("2010-04-05"))
 

```
[1] "Created Monday, Apr 05, 2010 00:00"
```

**Tip:** use a date with day > 12

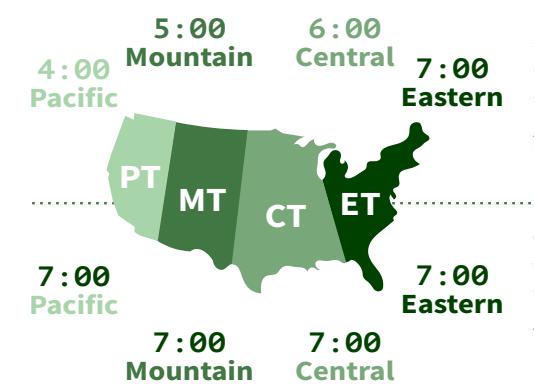
# Time Zones

R recognizes ~600 time zones. Each encodes the time zone, Daylight Savings Time, and historical calendar variations for an area. R assigns one time zone per vector.

Use the **UTC** time zone to avoid Daylight Savings.

**OlsonNames()** Returns a list of valid time zone names. OlsonNames()

**Sys.timezone()** Gets current time zone.



**with\_tz(time, tzzone = "")** Get the same date-time in a new time zone (a new clock time). Also **local\_time(dt, tz, units)**. with\_tz(dt, "US/Pacific")

**force\_tz(time, tzzone = "")** Get the same clock time in a new time zone (a new date-time). Also **force\_tzs()**. force\_tz(dt, "US/Pacific")

Math with date-times relies on the **timeline**, which behaves inconsistently. Consider how the timeline behaves during:

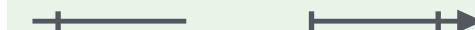
#### A normal day

```
nor <- ymd_hms("2018-01-01 01:30:00", tz="US/Eastern")
```



The start of daylight savings (spring forward)

```
gap <- ymd_hms("2018-03-11 01:30:00", tz="US/Eastern")
```



The end of daylight savings (fall back)

```
lap <- ymd_hms("2018-11-04 00:30:00", tz="US/Eastern")
```



Leap years and leap seconds

```
leap <- ymd("2019-03-01")
```



**Periods** track changes in clock times, which ignore time line irregularities.

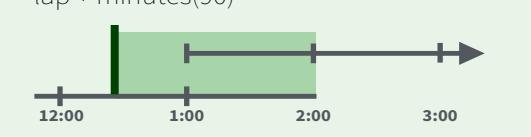
nor + minutes(90)



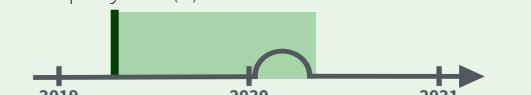
gap + minutes(90)



lap + minutes(90)

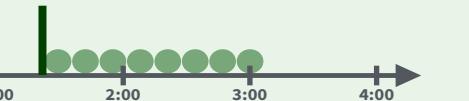


leap + years(1)



**Durations** track the passage of physical time, which deviates from clock time when irregularities occur.

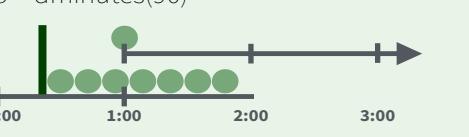
nor + dminutes(90)



gap + dminutes(90)



lap + dminutes(90)

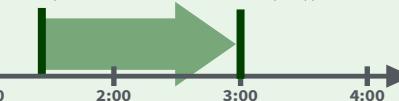


leap + dyears(1)



**Intervals** represent specific intervals of the timeline, bounded by start and end date-times.

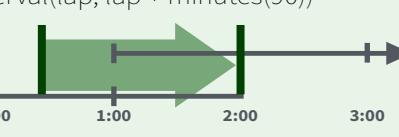
interval(nor, nor + minutes(90))



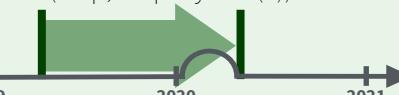
interval(gap, gap + minutes(90))



interval(lap, lap + minutes(90))



interval(leap, leap + years(1))



Not all years are 365 days due to **leap days**.

Not all minutes are 60 seconds due to **leap seconds**.

It is possible to create an imaginary date by adding **months**, e.g. February 31st

```
jan31 <- ymd("2018-01-31")
```

```
jan31 + months(1)
```

```
NA
```

**%m+m%** and **%m-%** will roll imaginary dates to the last day of the previous month.

```
jan31 %m+% months(1)
```

```
"2018-02-28"
```

**add\_with\_rollback(e1, e2, roll\_to\_first = TRUE)** will roll imaginary dates to the first day of the new month.

```
add_with_rollback(jan31, months(1), roll_to_first = TRUE)
```

```
"2018-03-01"
```

## PERIODS

Add or subtract periods to model events that happen at specific clock times, like the NYSE opening bell.

Make a period with the name of a time unit **pluralized**, e.g.

```
p <- months(3) + days(12)
```

"3m 12d 0H 0M 0S"

Number of months Number of days etc.

**years(x = 1)** x years.

**months(x = 1)** x months.

**weeks(x = 1)** x weeks.

**days(x = 1)** x days.

**hours(x = 1)** x hours.

**minutes(x = 1)** x minutes.

**seconds(x = 1)** x seconds.

**milliseconds(x = 1)** x milliseconds.

**microseconds(x = 1)** x microseconds.

**nanoseconds(x = 1)** x nanoseconds.

**picoseconds(x = 1)** x picoseconds.

**period(num = NULL, units = "second", ...)**

An automation friendly period constructor.  
period(5, unit = "years")

**as.period(x, unit)** Coerce a timespan to a period, optionally in the specified units.  
Also **is.period()**. as.period(i)

**period\_to\_seconds(x)** Convert a period to the "standard" number of seconds implied by the period. Also **seconds\_to\_period()**.  
period\_to\_seconds(p)

## DURATIONS

Add or subtract durations to model physical processes, like battery life. Durations are stored as seconds, the only time unit with a consistent length.

**Difftimes** are a class of durations found in base R.

Make a duration with the name of a period prefixed with a **d**, e.g.

```
dd <- ddays(14)
```

```
dd
```

"1209600s (~2 weeks)"

Exact length in seconds  
Equivalent in common units

**dyears(x = 1)** 31536000x seconds.

**dmonths(x = 1)** 2629800x seconds.

**dweeks(x = 1)** 604800x seconds.

**ddays(x = 1)** 86400x seconds.

**dhours(x = 1)** 3600x seconds.

**dminutes(x = 1)** 60x seconds.

**dseconds(x = 1)** x seconds.

**dmilliseconds(x = 1)** x × 10<sup>-3</sup> seconds.

**dmicroseconds(x = 1)** x × 10<sup>-6</sup> seconds.

**dnanoseconds(x = 1)** x × 10<sup>-9</sup> seconds.

**dpicoseconds(x = 1)** x × 10<sup>-12</sup> seconds.

**duration(num = NULL, units = "second", ...)**

An automation friendly duration constructor. duration(5, unit = "years")

**as.duration(x, ...)** Coerce a timespan to a duration. Also **is.duration()**, **is.difftime()**. as.duration(i)

**make\_difftime(x)** Make difftime with the specified number of units.  
make\_difftime(99999)

## INTERVALS

Divide an interval by a duration to determine its physical length, divide an interval by a period to determine its implied length in clock time.

Make an interval with **interval()** or **%--%**, e.g.

```
i <- interval(ymd("2017-01-01"), d)
```

```
j <- d %--% ymd("2017-12-31")
```

```
2017-01-01 UTC--2017-11-28 UTC
```

```
2017-11-28 UTC--2017-12-31 UTC
```

Start Date End Date



a **%within%** b Does interval or date-time a fall within interval b? now() %within% i



**int\_start(int)** Access/set the start date-time of an interval. Also **int\_end()**. int\_start(i) <- now(); int\_start(i)



**int\_aligns(int1, int2)** Do two intervals share a boundary? Also **int\_overlaps()**. int\_aligns(i, j)



**int\_diff(times)** Make the intervals that occur between the date-times in a vector.  
v <- c(dt, dt + 100, dt + 1000); int\_diff(v)



**int\_flip(int)** Reverse the direction of an interval. Also **int\_standardize()**. int\_flip(i)



**int\_length(int)** Length in seconds. int\_length(i)



**int\_shift(int, by)** Shifts an interval up or down the timeline by a timespan. int\_shift(i, days(-1))



**as.interval(x, start, ...)** Coerce a timespan to an interval with the start date-time. Also **is.interval()**. as.interval(days(1), start = now())

# Machine Learning Modelling in R : : CHEAT SHEET

## Supervised & Unsupervised Learning

| ALGORITHM                           | DESCRIPTION                                                                                                                                                                                                                                                      | R PACKAGE::FUNCTION                                                                | SAMPLE CODE                                                                                                                                                                                                                                                                                 |
|-------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NBC<br>Naïve Bayes classifier       | A classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naïve Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature | e1071::naiveBayes                                                                  | naiveBayes(class ~ ., data = x)                                                                                                                                                                                                                                                             |
| KNN<br>k-Nearest Neighbours         | A non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression                           | class::knn                                                                         | knn(train, test, cl, k = 1, l = 0, prob = FALSE, use.all = TRUE)                                                                                                                                                                                                                            |
| REG<br>Linear Regression            | Model the linear relationship between a scalar dependent variable Y and one or more explanatory variables (or independent variables) denoted X                                                                                                                   | stats::lm                                                                          | lm(dist ~ speed, data=cars)                                                                                                                                                                                                                                                                 |
| LREG<br>Logistic Regression         | Used to predict a binary outcome (1 / 0, Yes / No, True / False) given a set of independent variables.                                                                                                                                                           | stats::glm                                                                         | glm(Y ~ ., family = binomial (link = 'logit'), data = X)                                                                                                                                                                                                                                    |
| TM<br>Tree-Based Models             | The idea is to consecutively divide (branch) the training dataset based on the input features until an assignment criterion with respect to the target variable into a "data bucket" (leaf) is reached                                                           | rpart::rpart                                                                       | rpart(Kyphosis ~ Age + Number + Start, data = kyphosis)                                                                                                                                                                                                                                     |
| ANN<br>Artificial Neural Network    | Neural networks are built from units called perceptrons. Perceptrons have one or more inputs, an activation function and an output. An ANN model is built up by combining perceptrons in structured layers.                                                      | neuralnet::neuralnet                                                               | neuralnet(f,data=train_,hidden=c(5,3),linear.output=T)                                                                                                                                                                                                                                      |
| SVM<br>Support Vector Machine       | A data classification method that separates data using hyperplanes                                                                                                                                                                                               | e1071::svm                                                                         | svm(formula, data = NULL, ..., subset, na.action = na.omit, scale = TRUE)                                                                                                                                                                                                                   |
| PCA<br>Principal Component Analysis | A procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.                                                        | stats::prcomp<br>stats::princomp<br>FactoMineR::PCA<br>ade4::dudi.pca<br>amap::acp | stats : prcomp(formula, data = NULL, subset, na.action, ...) stats : princomp(formula, data = NULL, subset, na.action, ...) FactoMineR : PCA(decathlon, quanti.sup = 11:12, quali.sup=13) ade4 : dudi.pca(deug\$stab, center = deug\$cent, scale = FALSE, scan = FALSE) amap : acp(lubisch) |
| kMC<br>k-Mean Clustering            | Aims at partitioning n observations into k clusters in which each observation belongs to the cluster with the nearest mean                                                                                                                                       | stats::kmeans                                                                      | kmeans(x, centers, iter.max = 10, nstart = 1, algorithm = c("Hartigan-Wong", "Lloyd", "Forgy", "MacQueen"), trace=FALSE)                                                                                                                                                                    |
| HCL<br>Hierarchical Clustering      | An approach which builds a hierarchy from the bottom-up, and doesn't require the number of clusters to be specified beforehand.                                                                                                                                  | stats::hclust                                                                      | hclust(d, method = "complete", members = NULL)                                                                                                                                                                                                                                              |

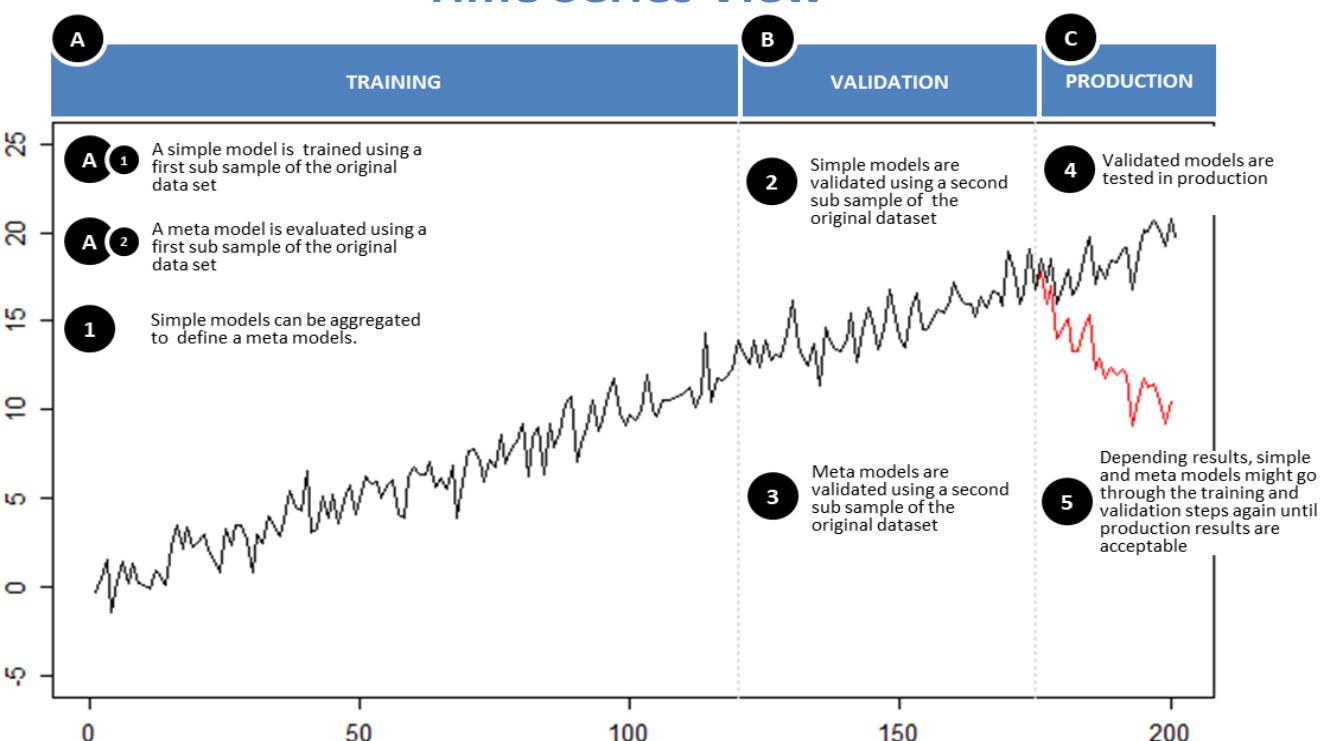
## Meta-Algorithm, Time Series & Model Validation

| ALGORITHM                                       | DESCRIPTION                                                                                                                                                                                                                                                           | R PACKAGE::FUNCTION                                                                                | SAMPLE CODE                                                                                                                                                                                                                               |
|-------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| REGU<br>Regularisation L1 (Lasso)<br>L2 (Ridge) | Regularization adds a penalty on the different parameters of a model to reduce the freedom of the model. Hence, the model will be less likely to fit the noise of the training data and will improve the generalization abilities of the model                        | glmnet::glmnet                                                                                     | L1 : glmnet(myMatrixA, myMatrixB , family = "gaussian", alpha = 1)<br>L2 : glmnet(myMatrixA, myMatrixB , family = "gaussian", alpha = 0)                                                                                                  |
| BOO<br>Boosting                                 | A process of iteratively refining, e.g. by reweighting, of estimated regression and classification functions (though it has primarily been applied to the latter), in order to improve predictive ability.                                                            | Parametric model - mboost::mboost                                                                  | glmboost(Y~ ., data = curr1\$trnidxs,)                                                                                                                                                                                                    |
| BAG<br>Bagging                                  | Bagging is a way to increase the power of a predictive statistical model by taking multiple random samples (with replacement) of the training data set, and using each of them to construct a separate model and separate predictions for the original test set       | All models: foreach Tree models: ipred::ipredbagging                                               | foreach : d <- data.frame(x=1:10, y=rnorm(10)) s <- foreach(d=iter(d, by="row"), .combine=rbind) %dopar%{ identical(s, d) ipred : bagging(formula, data, subset, na.action=na.rpart, .dots)}                                              |
| PRU<br>Pruning                                  | Pruning is a technique that reduces the size of decision tree by removing sections of the tree that provide little power to classify instances. Pruning reduces the complexity of the final classifier and hence improves predictive accuracy by reducing overfitting | rpart::rpart                                                                                       | prune(x, cp = 0.1)                                                                                                                                                                                                                        |
| RFO<br>Random Forest                            | An ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression)     | randomForest::randomForest                                                                         | randomForest(X ~ ., data = Y, subset = mySub)                                                                                                                                                                                             |
| STS<br>Time Series                              | Lead-lag analysis, Auto-correlation, Spectral analysis, Time series clustering, Seasonality, Trend....                                                                                                                                                                | stats xts forecast spectral TTR ....                                                               | Auto-correlation: acf(x, lag.max = NULL, type = c("correlation", "covariance", "partial")) Spectral Analysis: spec.pgram(myTs, spans = NULL) Seasonal Decomposition of Time Series - stl(x, s.window = 7, t.window = 50, t.jump = 1) .... |
| PM<br>Performance metrics                       | Depends on the problem:<br>• Regression: squared errors, outliers, error rate...<br>• Classification: Accuracy, precision, recall, F-score...                                                                                                                         | Regression-stats::outlierTest, stats::qqPlot ... Classification-ROCR::Tree: caret::confusionMatrix | Regression: fit <- lm(Y~X,data=myData) outlierTest(fit) qqPlot(fit, main="QQ Plot")                                                                                                                                                       |
| BVT<br>Bias-Variance Tradeoff                   | Simple models with few parameters are easier to compute but may lead to poorer fits (high bias). Complex models may provide more accurate fits but may over-fit the data (high variance)                                                                              | Tailored to the analysis                                                                           | Tailored to the analysis                                                                                                                                                                                                                  |
| CV<br>Cross validation                          | Cross validation compares the test performances of different model realisations with different sets or values of parameters                                                                                                                                           | caret::createDataPartition caret::createFolds                                                      | createDataPartition(classes, p = 0.8, list = FALSE)                                                                                                                                                                                       |
| LC<br>Learning Curves                           | Learning curves plot a model's training and test errors, or the chosen performance metric, depending on the training set size                                                                                                                                         | caret::learning_curve_dat                                                                          | learning_curve_dat(dat,outcome = NULL, proportion = (1:10)/10, test_prop = 0, verbose = TRUE, ...)                                                                                                                                        |

## Standard Modelling Workflow



## Time Series View



# Machine Learning with R



## Introduction

**mlr** offers a unified interface for the basic building blocks of machine learning: **tasks**, **learners**, **hyperparameters**, etc.

**Tasks** contain a description of a task (classification, regression, clustering, etc.) and a data set.

**Learners** specify a machine learning algorithm (GLM, SVM, xgboost, etc.) and its parameters.

**Hyperparameters** are learner settings that can be specified directly or tuned. A **parameter set** lists the possible hyperparameters for a given learner.

**Wrapped Models** are learners that have been trained on a task and can be used to make predictions.

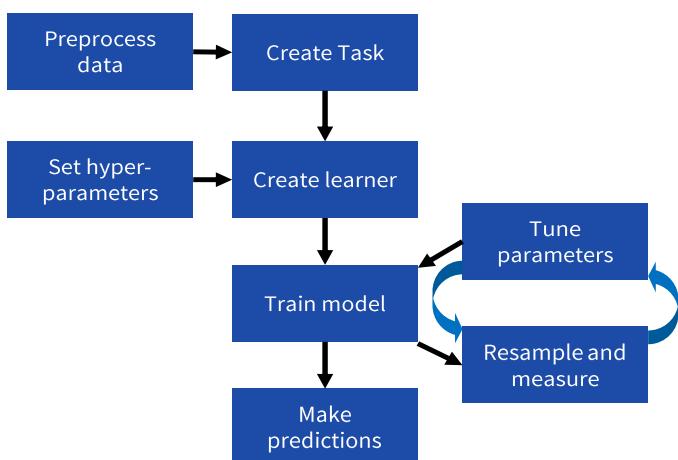
**Predictions** are the results of applying a model to either new data or the original training data.

**Measures** control how learner performance is evaluated, e.g. RMSE, LogLoss, AUC, etc.

**Resampling** estimates generalization performance by separating training data from test data. Common strategies include holdout and cross-validation.

Links: [Tutorial](#) | [CRAN](#) | [Github](#)

## mlr workflow



## Setup

### Preprocessing data

`createDummyFeatures(obj=, target=, method=, cols=)`  
Creates (0,1) flags for each non-numeric variable excluding `target`. Can be applied to entire dataset or only specific `cols`

`normalizeFeatures(obj=, target=, method=, cols=, range=, on.constant=)`  
Normalizes numerical features according to specified `method`:

- "center" (subtract mean)
- "scale" (divide by std. deviation)
- "standardize" (center and scale)
- "range" (linear scale to given range, default `range=c(0,1)`)

`mergeSmallFactorLevels(task=, cols=, min.perc=)`  
Combine infrequent factor levels into a single merged level

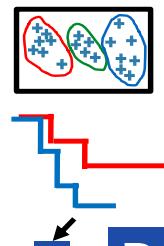
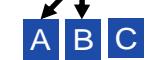
`summarizeColumns(obj=)` where `obj` is a data.frame or task. Provides type, NA, and distributional data about each column

See also `capLargeValues` `dropFeatures`  
`removeConstantFeatures` `summarizeLevels`

### Creating a task



0 63 100



`makeClassifTask(data=, target=)`  
Classification of a target variable, with optional positive class `positive`

`makeRegrTask(data=, target=)`  
Regression on a target variable

`makeMultilabelTask(data=, target=)`  
Classification where the target can belong to more than one class per observation

`makeClusterTask(data=)`  
Unsupervised clustering on a data set

`makeSurvTask(data=, target= c("time", "event"))`  
Survival analysis with a survival time column and an event column

`makeCostSensTask(data=, costs=)`  
Cost-sensitive classification where each observation-cost pair has a specified cost

Other arguments that can be passed to a `task`:

- `weights`=Weighting vector to apply to observations
- `blocking`=Factor vector where each level indicates a block of observations that will not be split up in resampling

### Making a learner

`makeLearner(cl=, predict.type=, ..., par.vals=)`  
Choose an algorithm class to perform the task and determine what that algorithm will predict

- `cl`=name of algorithm, e.g. `"classif.xgboost"` `"regr.randomForest"` `"cluster.kmeans"`
  - `predict.type="response"` returns a prediction type that matches the source data; `"prob"` returns a predicted probability for classification problems only; `"se"` returns the standard error of the prediction for regression problems only. Only certain learners can return `"prob"` and `"se"`
  - `par.vals`=takes a list of hyperparameters and passes them to the learner; parameters can also be passed directly (...)
- You can make multiple learners at once with `makeLearners()`

mlr has integrated over 170 different learning algorithms

- Full list: `View(listLearners())` shows all learners
- Available learners for a task: `View(listLearners(task))`
- Filtered list: `View(listLearners("classif", properties=c("prob", "factors")))` shows all classification learners `"classif"` which can predict probabilities `"prob"` and handle factor inputs `"factors"`
- See also `getLearnerProperties()`

## Training & Testing

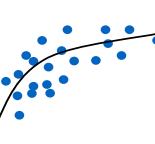
### Setting hyperparameters



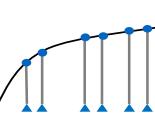
`setHyperPars(learner=, ...)`  
Set the hyperparameters (settings) for each learner, if you don't want to use the defaults. You can also specify hyperparameters in the `makeLearner()` call

`getParamSet(learner=)`  
Show the possible universe of parameters for your learner; can take a learner directly, or a text string such as `"classif.qda"`

### Train a model and predict



`train(learner=, task=)`  
Train a model (`WrappedModel`) by applying a learner to a task. By default, the model will train on all observations. The underlying model can be extracted with `getLearnerModel()`



`predict(object=, task=, newdata=)`  
Use a trained model to make predictions on a task or dataset. The resulting `pred` object can be viewed with `View(pred)` or accessed by `as.data.frame(pred)`

### Measuring performance

`performance(pred=, measures=)`  
Calculate performance of predictions according to one or more of several measures (use `listMeasures()` for full list):

- `classif` acc auc bac brier[,scaled] f1 fdr fn fnr fp fpr gmean multiclass[,au1u .aupn .abrier] npv ppv qsr rsr tn tnr tp tpr wkappa
- `regr` arsq expvar kendalltau mae mape medae medse mse msle rae rmse rmsle rrse rsq sae spearmanrho sse
- `cluster` db dunn G1 G2 silhouette
- `multilabel` multilabel[,f1 .subset01 .tpr .ppv .acc .hamloss]
- `costsens` mcp meancosts
- `surv` cindex
- `other` featperc timeboth timetrain

For detailed performance data on classification tasks, use:

- `calculateConfusionMatrix(pred=)`
- `calculateROCMeasures(pred=)`

### Resampling a learner

`makeResampleDesc(method=, ..., stratify=)`

`method` must be one of the following:

- "CV" (cross-validation, for number of folds use `iters=`)
- "LOO" (leave-one-out cross-validation, for folds use `iters=`)
- "RepCV" (repeated cross-validation, for number of repetitions use `reps=`, for folds use `folds=`)
- "Subsample" (aka Monte-Carlo cross-validation, for iterations use `iters=`, for train % use `split=`)
- "Bootstrap" (out-of-bag bootstrap, uses `iters=`)
- "Holdout" (for train % use `split=`)

`stratify` keeps target proportions consistent across samples.

`makeResampleInstance(desc=, task=)` can reduce noise by ensuring the resampling is done identically every time.

`resample(learner=, task=, resampling=, measures=)`  
Train and test model according to specified resampling strategy.

mlrincludes several pre-specified resample descriptions: `cv2` (2-fold cross-validation), `cv3`, `cv5`, `cv10`, `hout` (holdout with split 2/3 for training, 1/3 for testing). Convenience functions also exist to `resample()` with a specific strategy: `crossval()`, `repCV()`, `holdout()`, `subsample()`, `bootstrapOOB()`, `bootstrapB632()`, `bootstrapB632plus()`

## Refining Performance

### Tuning hyperparameters

Set search space using `makeParamSet(make<type>Param())`

- `makeNumericParam(id=, lower=, upper=, trafo=)`
- `makeIntegerParam(id=, lower=, upper=, trafo=)`
- `makeIntegerVectorParam(id=, len=, lower=, upper=, trafo=)`
- `makeDiscreteParam(id=, values=c(...))` (can also be used to test discrete values of numeric or integer parameters)
- `trafo` transforms the parameter output using a specified function, e.g. `lower=-2,upper=2,trafo=function(x) 10^x` would test values between 0.01 and 100, scaled exponentially
- Other acceptable parameter types include `Logical` `LogicalVector` `CharacterVector` `DiscreteVector`

Set a search algorithm with `makeTuneControl<type>()`

- `Grid(resolution=10L)` Grid of all possible points
- `Random(maxit=100)` Randomly sample search space
- `MBO(budget=)` Use Bayesian model-based optimization
- `Irace(n.instances=)` Iterated racing process
- Other types: `CMAES`, `Design`, `GensA`

Tune using `tuneParams(learner=, task=, resampling=, measures=, par.set=, control=)`

## Quickstart

### Prepare data for training and testing

```

library(mlbench)
data(Soybean)
soy = createDummyFeatures(Soybean,target="Class")
tsk = makeClassifTask(data=soy,target="Class")
ho = makeResampleInstance("Holdout",tsk)
tsk.train = subsetTask(tsk,ho$train.ind$[1])
tsk.test = subsetTask(tsk,ho$test.ind$[1])

```

Convert the factor inputs in the Soybean dataset into (0,1) dummy features which can be used by the XGboost algorithm. Create a task to predict the "Class" column. Create a train set with 2/3 of data and a test set with the remaining 1/3 (default).

### Create learner and evaluate performance

```

lrn = makeLearner("classif.xgboost",nrounds=10)
cv = makeResampleDesc("CV",iters=5)
res = resample(lrn,tsk.train,cv,acc)

```

Create an XGboost learner which will build 10 trees. Then test performance using 5-fold cross-validation. Accuracy should be between 0.90-0.92.

### Tune hyperparameters and retrain model

```

ps = makeParamSet(makeNumericParam("eta",0,1),
 makeNumericParam("lambda",0,200),
 makeIntegerParam("max_depth",1,20))
tc = makeTuneControlMBO(budget=100)
tr = tuneParams(lrn,tsk.train,cv5,acc,ps,tc)
lrn = setHyperPars(lrn,par.vals=tr$x)

```

Tune hyperparameters `eta`, `lambda`, and `max_depth` by defining a search space and using Model Based Optimization (MBO) to control the search. Then perform 100 rounds of 5-fold cross-validation, improving accuracy to ~0.93. Update the XGboost learner with the tuned hyperparameters.

```

mdl = train(lrn,tsk.train)
prd = predict(mdl,tsk.test)
calculateConfusionMatrix(prd)
mdl = train(lrn,tsk)

```

Train the model on the train set and make predictions on the test set. Show performance as a confusion matrix. Finally, re-train model on the full set to use on new data. You are now ready to go out into the real world and make 93% accurate predictions!

Legend for functions (not all parameters shown):

`function(required_parameters=,optional_parameters=)`

## Configuration

mlr's default settings can be changed using `configureMlr()`:

- `show.info` Whether to show verbose output by default when training, tuning, resampling, etc. (`TRUE`)
- `on.learner.error` How to handle a learner error. "`stop`" halts execution, "`warn`" returns NAs and displays a warning, "`quiet`" returns NAs with no warning ("`stop`")
- `on.learner.warning` How to handle a learner warning. "`warn`" displays a warning, "`quiet`" suppresses it ("`warn`")
- `on.par.without.desc` How to handle a parameter with no description. "`stop`", "`warn`", "`quiet`" ("`stop`)
- `on.par.out.of.bounds` How to handle a parameter with an out-of-bounds value. "`stop`", "`warn`", "`quiet`" ("`stop`)
- `on.measure.not.applicable` How to handle a measure not applicable to a learner. "`stop`", "`warn`", "`quiet`" ("`stop`)
- `show.learner.output` Whether to show learner output to the console during training (`TRUE`)
- `on.error.dump` Whether to create an error dump for crashed learners if `on.learner.error` is not set to "`stop`" (`TRUE`)

Use `getMlrOptions()` to see current settings

## Parallelization

mlr works with the `parallelMap` package to take advantage of multicore and cluster computing for faster operations. mlr automatically detects which operations are able to run in parallel.

To begin parallel operation use:  
`parallelStart(mode=, cpus=, level=)`

- `mode` determines how the parallelization is performed:
  - "`local`" no parallelization applied, simply uses `mapply`
  - "`multicore`" multicore execution on a single machine, uses `parallel::mclapply`. Not available in Windows.
  - "`socket`" multicore execution in socket mode
  - "`mpi`" Snow MPI cluster on one or multiple machines using `parallel::makeCluster` and `parallel::clusterMap`
  - "`BatchJobs`" Batch queuing HPC clusters using `BatchJobs::batchMap`
- `cpus` determines how many logical cores will be used
- `level` controls parallelization: "`mlr.benchmark`", "`mlr.resample`" "`mlr.selectFeatures`", "`mlr.tuneParams`" "`mlr.ensemble`"

To end parallelization, use `parallelStop()`

## Imputation

`impute(obj=, target=, cols=, dummy.cols=, dummy.type=)`  
 Applies specified logic to data frame or task containing NAs and returns an imputation description which can be used on new data

- `obj`=data frame or task on which to perform imputation
- `target`=specify target variable which will not be imputed
- `cols`=column names and logic for imputation\*
- `dummy.cols`=column names to create a NA (T/F) column\*
- `dummy.type`=set to "`numeric`" to use (0,1) instead of (T/F)\*

\*Can also use `classes` and `dummy.classes` in place of `cols`

Imputation logic is passed to `cols` or `classes` via a list, e.g.: `cols=list(V1=imputeMean())` where `V1` is the column to which to apply the imputation, and `imputeMean()` is the imputation method. Available imputation methods include:

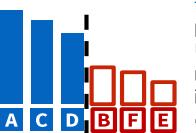
```
imputeConst(const=) imputeMedian() imputeMode()
imputeMin(multiplier=) imputeMax(multiplier=)
imputeNormal(mean=, sd=)
imputeHist(breaks=, use.mids=)
imputeLearner(learner=, features=)
impute
```

`impute` returns a list containing the imputed dataset or task as well as an imputation description that can be used to reapply the same imputation to new data using `reimpute`

`reimpute(obj=, desc=)` Imputes missing values on a task or dataset (`obj`) using a description (`desc`) created by `impute`

## Feature Extraction

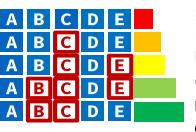
### Feature filtering



`filterFeatures(task=, method=, perc=, abs=, threshold=)`  
 Uses a learner-agnostic feature evaluation method to rank feature importance, then includes only features in the top n percent (`perc=`), top n (`abs=`), or which meet a set performance threshold (`threshold=`).

Outputs a task with features that failed the test omitted. `method` defaults to "`randomForestSRC.rfsrc`", but can be set to: "`anova.test`" "`carscore`" "`cforest.importance`" "`chi.squared`" "`gain.ratio`" "`information.gain`" "`kruskal.test`" "`linear.correlation`" "`mrmr`" "`oneR`" "`permutation.importance`" "`randomForest.importance`" "`randomForestSRC.rfsrc`" "`randomForestSRC.var.select`" "`rank.correlation`" "`relief`" "`symmetrical.uncertainty`" "`univariate.model.score`" "`variance`"

### Feature selection



`selectFeatures(learner=, task=, resampling=, measures=, control=)`  
 Uses a feature selection algorithm (`control`) to resample and build a model repeatedly using different feature sets each time in order to find the best set.

Available controls include:

- `makeFeatSelControlExhaustive(max.features=)` Try every combination of features up to optional `max.features`
- `makeFeatSelControlRandom(maxit=, prob=, max.features=)` Randomly sample features with probability `prob` (default 0.5) until `maxit` (default 100) iterations; return the best one found
- `makeFeatSelControlSequential(method=, maxit=, max.features=, alpha=, beta=)` Perform an iterative search using a `method` from the following: "`sfs`" forward search, "`sbs`" backward search, "`sffs`" floating forward search, "`sfbs`" floating backward search. `alpha` indicates minimum improvement required to add a feature; `beta` indicates minimum required to remove a feature
- `makeFeatSelControlGA(maxit=, max.features=, mu=, lambda=, crossover.rate=, mutation.rate=)` Genetic algorithm trains on random feature vectors, then uses crossover on the best performers to produce 'offspring', repeated over generations. `mu` is size of parent population, `lambda` is size of children population, `crossover.rate` is probability of choosing a bit from first parent, `mutation.rate` is probability of flipping a bit (on or off)

`selectFeatures` returns a `FeatSelResult` object which contains optimal features and an optimization path. To apply feature selection result (`fsr`) to your task (`tsk`), use:  
`tsk = subsetTask(tsk, features=fsr$x)`

## Benchmarking

`benchmark(learners=, tasks=, resamplings=, measures=)`  
 Allows easy comparison of multiple learners on a single task, a single learner on multiple tasks, or multiple learners on multiple tasks. Returns a benchmark result object.

Benchmark results can be accessed with a variety of functions beginning with `getBMR<object>: AggrPerformance`  
`FeatSelResults` `FilteredFeatures` `LearnerIds`  
`LeanerShortNames` `Learners` `MeasureIds` `Measures`  
`Models` `Performances` `Predictions` `TaskDescs` `TaskIds`  
`TuneResults`

mlr contains several toy tasks which are useful for benchmarking:  
`agri.task` `bc.task` `bh.task` `costiris.task` `iris.task`  
`lung.task` `mtcars.task` `pid.task` `sonar.task`  
`wpbc.task` `yeast.task`

## Visualization

### Performance

`generateThreshVsPerfData(obj=, measures=)` Measure performance at different probability cutoffs to determine optimal decision threshold for binary classification problems

- `plotThreshVsPerf(obj)` Plot visual representation of threshold curve(s) from `ThreshVsPerfData`
- `plotROCCurves(obj)` Plot receiver operating characteristic (ROC) curve from `ThreshVsPerfData`. Must set `measures=list(fpr,tpr)`

### Residuals

- `plotResiduals(obj)` Plots residuals for `Prediction` or `BenchmarkResult`

### Learning curve

`generateLearningCurveData(learners=, task=, resampling=, percs=, measures=)` Measure performance of learner(s) trained on different percentages of task data

- `plotLearningCurve(obj)` Plot curve showing learner performance vs. proportion of data used, uses `LearningCurveData`

### Feature importance

`generateFilterValuesData(task=, method=)` Get feature importance rankings using specified filter method

- `plotFilterValues(obj)` Plot bar chart of feature importance based on filter method using `FilterValuesData`

### Hyperparameter tuning

`generateHyperParsEffectData(tune.result=)` Get the impact of different hyperparameter settings on model performance

- `plotHyperParsEffect(hyperpars.effec.t.data=, x=, y=, z=)` Create a plot showing hyperparameter impact on performance using `HyperParsEffectData`

See also:

- `plotOptPath(op=)` Display details of optimization process. Takes `<obj>$opt.path`, where `<obj>` is an object of class `tuneResult` or `featSelResult`
- `plotTuneMultiCritResult(res=)` Show pareto front for results of tuning to multiple performance measures

### Partial dependence

`generatePartialDependenceData(obj=, input=)` Get partial dependence of model (`obj`) prediction over each feature of data (`input`)

- `plotPartialDependence(obj)` Plots partial dependence of model using `PartialDependenceData`

### Benchmarking

- `plotBMRBoxplots(bmr=)` Distribution of performances
- `plotBMRSummary(bmr=)` Scatterplot of avg. performances
- `plotBMRRanksAsBarChart(bmr=)` Rank learners in bar plot

### Other

- `generateCritDifferencesData(bmr=, measures=, p.value=, test=)` Perform critical-differences test using either the Bonferroni-Dunn ("bd") or "Nemenyi" test
- `plotCritDifferences(obj=)`

- `generateCalibrationData(obj=)` Evaluate calibration of probability predictions vs. true incidence
- `plotCalibration(obj=)`

## Wrappers

**Wrappers** fuse a learner with additional functionality. mlr treats a learner with wrappers as a single learner, and hyperparameters of wrappers can be tuned jointly with underlying model parameters. Models trained with wrappers will apply them to new data.

### Preprocessing and imputation

`makeDummyFeaturesWrapper(learner=)`  
`makeImputeWrapper(learner=, classes=, cols=)`  
`makePreprocWrapper(learner=, train=, predict=)`  
`makePreprocWrapperCaret(learner=, ...)`  
`makeRemoveConstantFeaturesWrapper(learner=)`

### Class imbalance

`makeOverBaggingWrapper(learner=)`  
`makeSMOTEWrapper(learner=)`  
`makeUndersampleWrapper(learner=)`  
`makeWeightedClassesWrapper(learner=)`

### Cost-sensitive learning

`makeCostSensClassifWrapper(learner=)`  
`makeCostSensRegrWrapper(learner=)`  
`makeCostSensWeightedPairsWrapper(learner=)`

### Multilabel classification

`makeMultilabelBinaryRelevanceWrapper(learner=)`  
`makeMultilabelClassifierChainsWrapper(learner=)`  
`makeMultilabelDBRWrapper(learner=)`  
`makeMultilabelNestedStackingWrapper(learner=)`  
`makeMultilabelStackingWrapper(learner=)`

### Other

`makeBaggingWrapper(learner=)`  
`makeConstantClassWrapper(learner=)`  
`makeDownsampleWrapper(learner=, dw.perc=)`  
`makeFeatSelWrapper(learner=, resampling=, control=)`  
`makeFilterWrapper(learner=, fw.perc=, fw.abs=, fw.threshold=)`  
`makeMultiClassWrapper(learner=)`  
`makeTuneWrapper(learner=, resampling=, par.set=, control=)`

## Nested Resampling

mlr supports **nested resampling** for complex operations such as tuning and feature selection through wrappers. In order to get a good estimate of generalization performance and avoid data leakage, both an outer (for tuning/feature selection) and an inner (for the base model) resampling process are advised.

- Outer resampling can be specified in `resample` or `benchmark`
- Inner resampling can be specified in `makeTuneWrapper`, `makeFeatSelWrapper`, etc.

## Ensembles

`makeStackedLearner(base.learners=, super.learner=, method=)` Combines multiple learners to create an ensemble

- `base.learners`=learners to use for initial predictions
- `super.learner`=learner to use for final prediction
- `method`=how to combine base learner predictions:
  - "average" simple average of all base learners
  - "stack.nocv", "stack.cv" train super learner on results of base learners, with or without cross-validation
  - "hill.climb" search for optimal weighted average
  - "compress" with a neural network for faster performance

## Specifying the Model

Possible syntaxes for specifying the variables in the model:

- nardl with fixed p and q lags**

```
nardl(fod~inf,p,q,data=fod,ic="aic",maxlags = FALSE,graph = FALSE,case=3)
```

- Auto selected lags (maxlags=TRUE)**

```
nardl(food~inf,data=fod,ic="aic",maxlags = TRUE,graph = FALSE,case=3)
```

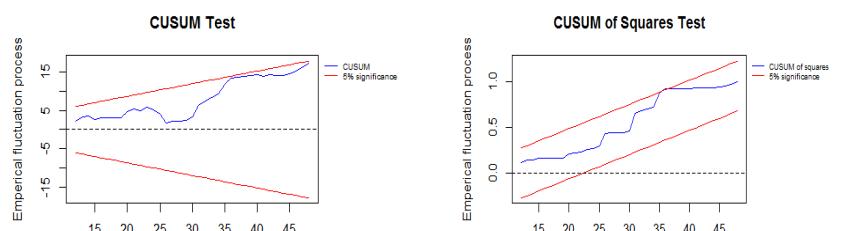
**The formula:**

- $y \sim x | z_1 + z_2 \dots$
- $y$  the dependent variable
- $x$  the decomposed variable (this package version can't assume more than one decomposed variable)
- $z_1 + z_2 + \dots$  independent variables
- Data** is the dataframe
- p** number of lags of the dependent variable
- q** number of lags of the independent variables
- ic** : c("aic", "bic", "II", "R2") criteria model selection
- maxlags** if **TRUE** auto lags selection
- case** case number 3 for (unrestricted intercept, no trend) and 5 (unrestricted intercept, unrestricted trend), 1 2 and 4 not supported

## Cusum and CusumQ plot

Cusum and CusumQ plot (graph=TRUE)

```
nardl(food~inf,data=fod,ic="aic",maxlags = TRUE,graph = TRUE,case=3)
```



## pssbounds specification include:

- Case case number 3 for (unrestricted intercept, no trend) and 5 (unrestricted intercept, unrestricted trend), 1 2 and 4 not supported
- fstat** represent the value of the F-statistic
- obs** represent the number of observation
- k** number of regressors appearing in lag levels

**Example:**

```
reg<-nardl(food~inf,fod,ic="aic",maxlags = TRUE,graph = TRUE,case=3)
pssbounds(case=reg$case,fstat=reg$fstat,obs=reg$obs,k=reg$k)
```

## LM test for serial correlation

LM test for serial correlation

```
bp2(object, nlags, fill = NULL, type = c("F", "Chi2"))
```

**Methods and options are:**

- object** fitted lm model
- nlags** positive integer number of lags
- fill** starting values for the lagged residuals in the auxiliary regression. By default 0.
- type** Fisher or Chisquare statistics

**Example :**

```
reg<-nardl(food~inf,fod,ic="aic",maxlags = TRUE,graph = TRUE,case=3)
```

```
bp2(regfit,regnp,fill=0,type="F")
```

## Lagrange multiplier test

Lagrange multiplier test for conditional heteroscedasticity of Engle (1982), as described by Tsay (2005, pp. 101-102)

```
ArchTest(x, lags = 12, demean = FALSE)
```

**Methods and options are:**

- x** numeric vector
- lags** positive integer number of lags.
- demean** logical: If TRUE, remove the mean before computing the test statistic.

**Example :**

```
reg<-nardl(food~inf,fod,ic="aic",maxlags = TRUE,graph = TRUE,case=3)
x<-reg$selresidu
nlag<-reg$np
ArchTest(x, lags=nlag)
```

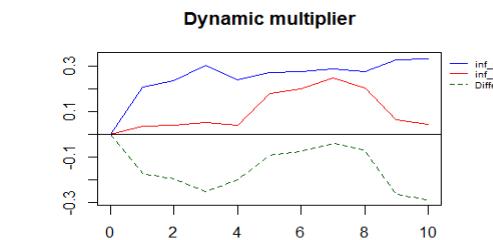
## plotmplier(model, np, k, h)

**Methods and options are:**

- model** the fitted model
- np** the selected number of lags
- k** number of decomposed independent variables
- h** is the horizon over which multipliers will be computed

**Example**

```
reg<-nardl(food~inf,p=4,q=4,fod,ic="aic",maxlags = FALSE,graph = TRUE,case=3)
plotmplier(reg,reg$np,1,10)
```



## pssbounds

pssbound function display the necessary critical values to conduct the Pesaran, Shin and Smith 2001 bounds test for cointegration. See <http://andyphilips.github.io/pssbounds/>.

```
pssbounds(obs, fstat, tstat = NULL, case, k)
```

**Methods and options are:**

- obs** number of observations
- fstat** value of the F-statistic
- tstat** value of the t-statistic
- case** case number
- k** number of regressors appearing in lag levels

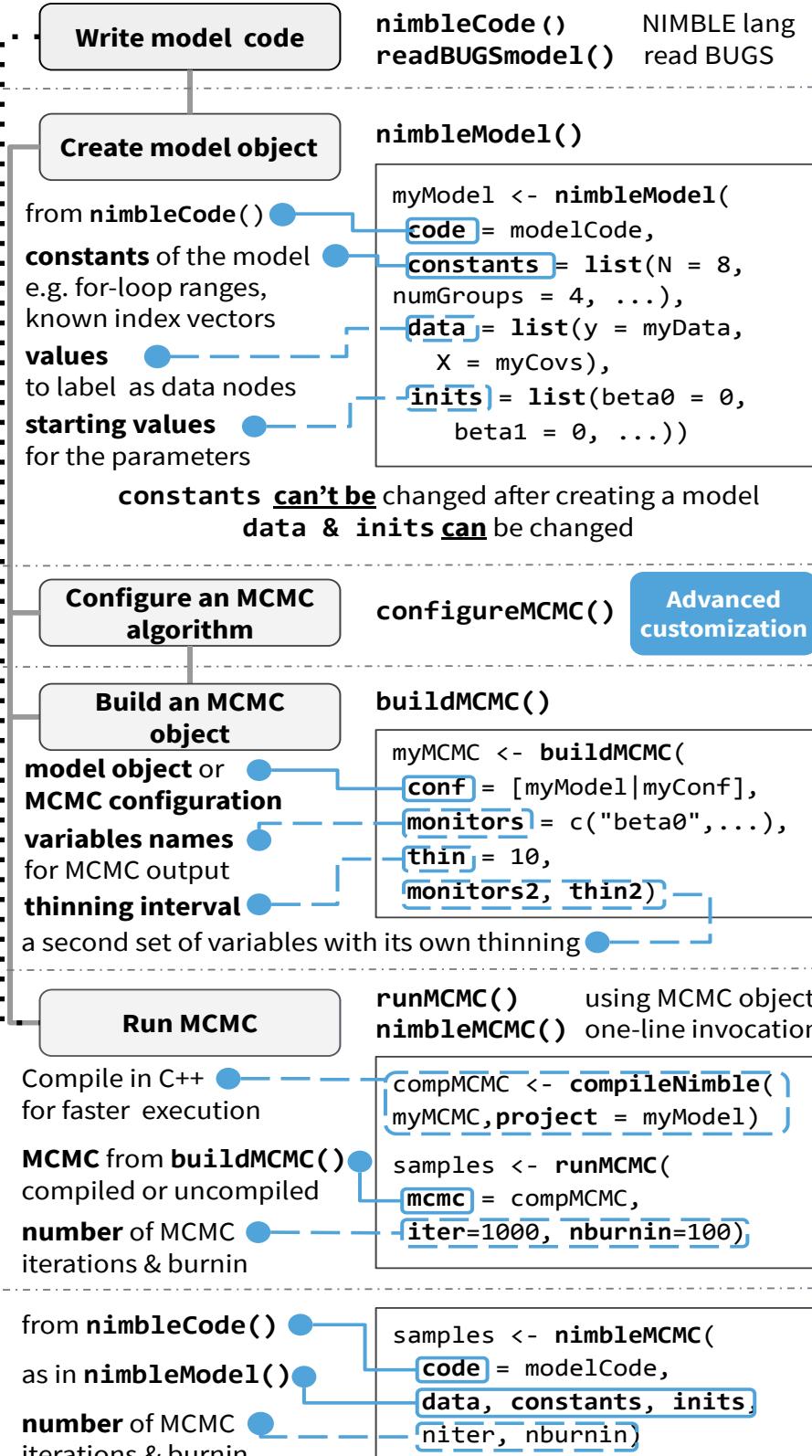
**Example**

```
reg<-nardl(food~inf,fod,ic="aic",maxlags = TRUE,graph = TRUE,case=3)
pssbounds(case=reg$case,fstat=reg$fstat,obs=reg$obs,k=reg$k)
F-stat concludes I(1) and cointegrating, t-stat concludes I(0).
```



# nimble models: : CHEAT SHEET

## NIMBLE workflow



required arguments  
optional arguments

## Writing model code

**Use named arguments** for non-default parameterization  
e.g. `beta0` and `beta1` follow equivalent distributions  
(default is precision, `tau`).

**Link functions** can be declared on the left-hand side.

**Order of declaration does not matter**  
`alpha[iGroup]` can be declared after being used in other declarations.

**Split code over multiple lines** to help people read it.

```
modelCode <- nimbleCode({
 beta0 ~ dnorm(0, sd = 1000)
 beta1 ~ dnorm(0, 1E-6)
 sdGroups ~ dunif(0, 100)
 fixed_effects[1:N] <- beta0 + beta1 * X[1:N]
 for(i in 1:N) {
 log(eta[i]) <- fixed_effects[i] +
 alpha[groupID[i]]
 y[i] ~ dpois(eta[i])
 }
 for(iGroup in 1:numGroups) {
 alpha[iGroup] ~ dnorm(0, sd = sdGroups)
 }
})
```

**Vectorized declarations** create vector nodes. This means `fixed_effects[1:N]` will be a single node. One vector node vs. multiple scalar nodes give different model graphs, so use with care.

**Provide explicit index ranges** or use empty brackets `([])` and provide the `dimensions` argument to `nimbleModel()`.

**Nested indexing** is a good way to implement experimental groups or factor levels. If groups are known from the design, include them in `constants`.

## Using models

**Models can be compiled.**  
`cModel <- compileNimble(myModel)`  
In methods below, "model" can be `cModel` or `myModel`.

**Models can access and change variables.**  
`model$beta0 <- 5`  
`model[["beta0"]] <- 5`

**Models can simulate or calculate log-probabilities.**

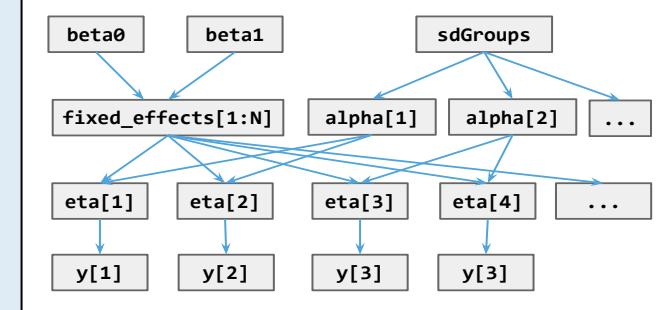
`model$calculate(nodes)`  
returns sum of log probability densities.

`model$calculateDiff(nodes)`  
returns difference in sum of log probability densities between current and previous node values.

`model$getLogProb(nodes)`  
returns sum of most recently calculated log probability densities.

`model$simulate(nodes, includeData = FALSE)`  
simulates into stochastic nodes.  
`includeData = FALSE` protects data.

### Models are graphs



**Models know about nodes, variables and relationships.**

`model$getNodeNames()`  
returns node names  
e.g. "eta[1]", "eta[2]", ...

`model$getVarNames()`  
returns variable names  
e.g. "eta"

`model$expandNodeNames(nodes)`  
e.g. "y" is expanded to "y[1]", "y[2]", ...

`model$getDependencies(nodes, ...)`  
returns nodes that depend on input nodes.

**Uncompiled models can be debugged, updated, and copied.**

**Flag nodes as data and set inits**  
`myModel$setData("y")`  
`myModel$setInits(inits)`  
**Debug model errors**  
`myModel$check()`  
check for missing/invalid values.  
`myModel$initializeInfo()`  
which nodes are not fully initialized?  
`myModel$checkBasics()`  
check for size/dimension mismatches and NA.  
**Make a copy**  
`myModel$newModel(replicate = TRUE)`

**Models know properties of nodes.**

`model$getDimension(node)`  
`model$getDistribution(nodes)`  
`model$isDeterm(nodes)`  
`model$isStoch(nodes)`  
`model$isData(nodes)`  
`model$isDiscrete(nodes)`  
`model$Multiivariate(nodes)`  
`model$Binary(nodes)`  
`model$EndNode(nodes)`  
`model$Truncated(nodes)`

# nimble distributions and functions: : CHEAT SHEET



## Declarations

```
STOCHASTIC x ~ ddist(args)
DETERMINISTIC z <- fn(args)
TRUNCATED STOCHASTIC x ~ T(ddist(args), min, max)
CENSORED STOCHASTIC seg ~ dinterval(t, c[1:nSegments])
t ~ ddist(args)
CONSTRAINT one ~ dconstraint(condition)
```

## Deterministic Functions

**SCALAR or COMPONENT-WISE**

**Logical:** |, &, !, >, >=, <, <=, !=, ==, equals, step

**Arithmetic:** +, -, \*, /, ^, pow(x, y)  
%%, exp, log, sqrt, abs, cube

**Trigonometric:** sin, cos, tan, asin, acos, atan, asinh, acosh, atanh

**Links:** logit, probit, cloglog  
(links can also be used on left-hand side of a declaration)

**Inverse links:** ilogit/expit, iprobit/phi, icloglog

**Rounding:** ceiling, floor, round, trunc

**Specials:** lgamma/loggam, besselK, log1p, lfactorial, logfact

**Distributions:** d, p, q, r forms of available distributions can be used as deterministic functions.

## VECTOR and/or MATRIX

**Returning scalar:** inprod, logdet, sum, mean, sd, prod, min, max

**Returning vector:** pmin, pmax, eigen(x)\$values, svd(x)\$d

**Returning matrix:** inverse, chol, %\*%, t, solve, forwardsolve, backsolve, eigen(x)\$vectors, svd(x)\$u, svd(x)\$v

## Write your own!

See Ch 12 of User Manual

NIMBLE allows you to write **new distributions and functions** using nimbleFunction().

## Univariate Distributions

### Continuous



#### BETA

y ~ dbeta([shape1, shape2 | mean, sd])  
shape1=mean^2\*(1-mean)/sd^2-mean  
shape2=mean\*(1-mean)^2/sd^2+mean-1



#### CHI-SQUARE

y ~ dchisq(df)



#### DOUBLE EXPONENTIAL (LAPLACE)

y ~ ddexp(location, [scale|rate|var])  
scale = 1/rate  
scale = sqrt(var/2)



#### EXPONENTIAL

y ~ dexp([rate|scale])  
rate = 1/scale



#### FLAT (improper)

y ~ dflat()



#### GAMMA

y ~ dgamma([shape, [rate|scale] | [mean, sd]])  
scale = 1/rate  
shape = mean^2/sd^2  
scale = sd^2/mean



#### HALF FLAT (improper)

y ~ dhalfflat()



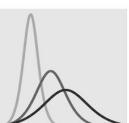
#### INVERSE GAMMA

y ~ dinvgamma(shape, [rate|scale])  
rate = 1/scale



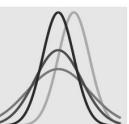
#### LOGISTIC

y ~ dlogis(location, [rate|scale])  
scale = 1/rate



#### LOG-NORMAL

y ~ dlnorm(meanlog, [taulog|sdlog|varlog])  
sdlog = 1/sqrt(taulog)  
sdlog = sqrt(varlog)



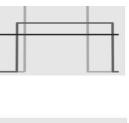
#### NORMAL

y ~ dnorm(mean, [tau|sd|var])  
sd = 1/sqrt(tau)  
sd = sqrt(var)



#### STUDENT T

y ~ dt(mu, [tau|sigma|sigma2], df)  
sigma = 1/sqrt(tau)  
sigma = sqrt(sigma2)



#### UNIFORM

y ~ dunif(min, max)



#### WEIBULL

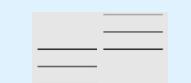
y ~ dweib(shape, [lambda|scale|rate])  
scale = lambda^(-1/shape)  
scale = 1/rate

## DISTRIBUTION NAME

y ~ ddist([default|alternative])  
canonical = fn(provided)

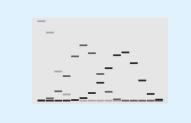
Lifted nodes are inserted when non-canonical parameters are used. Default parameters are not necessarily canonical.

## Discrete



#### BERNOULLI

y ~ dbern(prob)



#### BINOMIAL

y ~ dbinom(prob, size)



#### CATEGORICAL

y ~ dcat(prob)



#### NEGATIVE BINOMIAL

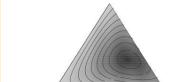
y ~ dnegbin(prob, size)



#### POISSON

y ~ dpois(lambda)

## Multivariate distributions



#### DIRICHLET

y[] ~ ddirch(alpha[])



#### MULTINOMIAL

y[] ~ dmulti(prob[], size)



#### MULTIVARIATE NORMAL

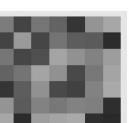
y[] ~ dmmnorm(mean[], [prec[,] | cov[,] | cholesky[,]], prec\_param)



#### MULTIVARIATE STUDENT T

y[] ~ dmvt(mu[], [prec[,] | scale[,] | cholesky[,]], df, prec\_param)

cholesky = chol(prec) : prec\_param=1  
cholesky = chol(cov) : prec\_param=0 for dmmnorm  
cholesky = chol(scale) : prec\_param=0 for dmvt  
cholesky is chol(prec) when prec\_param=1,  
chol(cov)|chol(scale) when prec\_param=0



#### WISHART

y[,] ~ dwish([R[,] | S[,] | cholesky[,]], df, scale\_param)



#### INVERSE WISHART

y[,] ~ dinvwish([S[,] | R[,] | cholesky[,]], df, scale\_param)

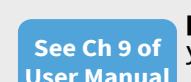
cholesky = chol(R): scale\_param=0  
cholesky = chol(S): scale\_param=1  
cholesky is chol(S) when scale\_param=1,  
chol(R) when scale\_param=0

## Distributions for spatial models



#### CONDITIONAL AUTOREGRESSIVE

intrinsic (improper)  
y[] ~ dcar\_normal(adj[], weights[], num[], tau, c, zero\_mean)



#### proper

y[] ~ dcar\_proper(mu[], C[], adj[], num[], M[], tau, gamma)



## Bayesian nonparametric distributions

See Ch 10 of User Manual



#### CHINESE RESTAURANT PROCESS

y[] ~ dCRP(conc, size)  
conc= concentration parameter



#### STICK BREAKING PROCESS

y[] ~ stick\_breaking(z[])  
z = vector of breaking points



# oSCR :: CHEAT SHEET

The oSCR package, pronounced “Oscar”, provides a set of functions for working with Spatial Capture Recapture (SCR) models.

## Getting the package

Package hosted on [GitHub](#)

```
library(devtools)
install_github("jaroyle/oSCR")
library(oSCR)
```

## Workflow

- Every model you run on oSCR has the following 4 basic steps.
- Modeled after [unmarked](#) workflow

### 1. Format the sampling data

- One file for each one:
- Spatial encounter histories
  - Detector information

### 2. Define and format the State Space

- Size and resolution of the state space
- Spatial covariates for density

### 3. Analyze the data – model fitting

- Likelihood based: use AIC to do model selection
- No need to use other packages, oSCR has helper functions to do the model selection.

### 4. Post processing model output for inference:

- This means that now that you have your parameters all you have to do is interpret your results!

## Modelling framework

### A. Single-session models

- Repeated sample occasions on a single population of individuals using a single array of traps.

### B. Multi-session models

- Data grouped in strata or groups which are independent in space or time.

### C. Explicit sex-structured models

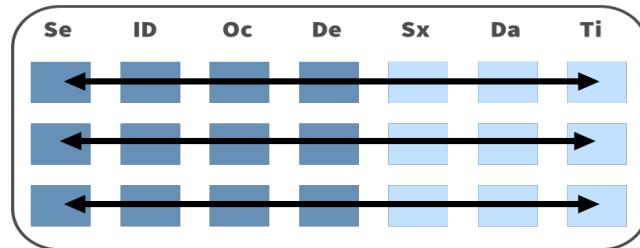
### D. Multi-session sex-structured models

## 1. Format sampling data

Before starting to use oSCR you need to format the datafiles in a scrFrame which consists of two basic spreadsheets: **edf** and **tdf**.

### 1.1 edf: encounter data file.

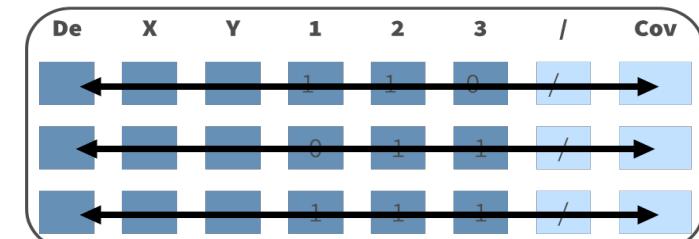
- Single **data frame**.
- Each row has individual detection events.
- Dark blue = required; light blue = optional.
- Columns contain capture information:
  - Session (Se)
  - Individual ID (ID)
  - Occasion (Oc)
  - Detector\* (De)
  - Sex (Sx)
  - Date (Da)
  - Time (Ti)



### 1.2 tdf: trap deployment data file.

- A **list** with information for each session (tdf1, tdf2, ...).
- Each row is a trap.
- Columns contain trap information
 

|                                                                                            |                                                          |
|--------------------------------------------------------------------------------------------|----------------------------------------------------------|
| - Detector* (De)                                                                           | 1, 2, 3, ... n                                           |
| - X (required, UTM)                                                                        | - Separator (e.g., /)                                    |
| - Y (required, UTM)                                                                        | - Trap level covariates (different column per covariate) |
| - Binary trap operation data for malfunctions, rotations (required if problems were found; |                                                          |



\*Notice that both edf and tdf have the same **Detector (De)** column that **MUST** match (same name, class, relational database).

**1.3 data2oscr()**: is a function that links **edf** and **tdf** files via the detector\* names. Creates **scrFrame**.

```
data <- data2oscr(
 edf, # encounter data file
 tdf, # list containing trap deployment file
 sess.col*, # session col number or name in edf
 id.col*, # individual ID col # or name in edf
 occ.col*, # occasion col number or name in edf
 trap.col*, # detector col number or name in edf
 sex.col*, # sex col number or name in edf
 sex.nancode, # character for unknown sex in edf
 K, # number of occasions
 ntraps, # number of traps
 trapcov.names, # vector of un-numbered cov names
 tdf.sep) # separator (e.g., "/")

* which(colnames(edf) %in% "name of column in edf")
```

### 1.4 Summary functions for scrFrame :

- scrFrame contains information from the **edf** and **tdf** via detector names.

```
sf<-data$scrFrame
```

**sf\$caphist** Array of individual-by-trap-by-occasion (n x J x K). Binary or counts.

**sf\$traps** Data frame containing at least trap ID and coordinates of traps. Best with UTM.

**sf\$indcovs** Sex data (0 female, 1 male) or any bivariate covariate. NAs allowed.

**sf\$trapCovs** List of session specific trap covariates. Row per trap, and column per covariate.

**sf\$sigCovs** A data frame of covariates that affect space use ( $\sigma$ ,  $\sigma$ ).

**sf\$trapOperation** A list of session specific information on trap operational data.

**sf\$occasions** A vector of number of occasions per session .

**sf\$mmdm** Mean maximum distance moved pooled across sessions.  $\frac{1}{2} mmdm \sim \sigma$

**sf\$mdm** Maximum distance moved pooled across sessions.

**\$telemetry** Telemetry object for fitting resource selection models.

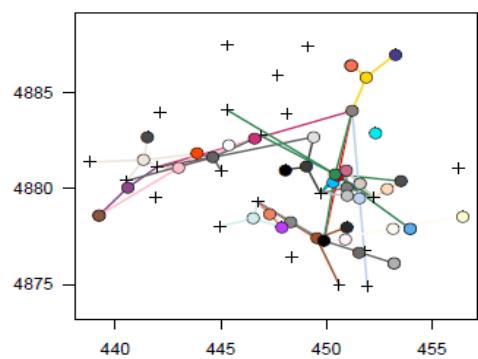
### 1.5 Summary of scrFrame

**sf**

|                  |      |
|------------------|------|
| S1               |      |
| n individuals    | 47   |
| n traps          | 38   |
| n occasions      | 8    |
| S1               |      |
| avg caps         | 3.21 |
| avg spatial caps | 2.02 |
| mmdm             | 4.65 |

### 1.6 Spatial captures per session

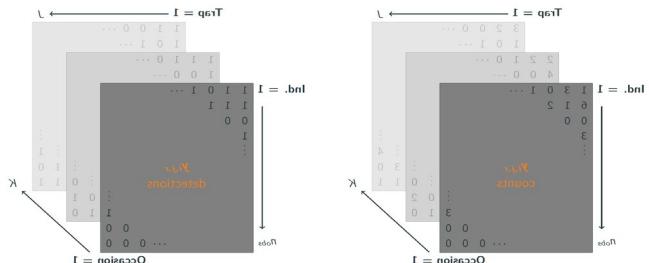
```
plot(sf) #y and x are UTM
```





# osCR :: CHEAT SHEET

## 1.4.1 Navigating the scrFrame



### Capture history

- Session 1, all individuals, all traps, occasion 3  
`sf$caphist[[1]][ , , 3]`
- Session 1, individual 4, all traps, all occasions  
`sf$caphist[[1]][4, , ]`

### Traps

- Session 1 trap coordinates  
`sf$traps[[1]]`

### Trap covariates

- Trap covariate df session 1 occasion 4  
`sf$trapCovs[[1]][[4]]`

### Trap operation

- Session 1 trap trap operation matrix  
`sf$trapOperation [[1]]`

### Covariates that affect sigma ( $\sigma$ )

- These covariates are NOT session specific.  
This is a sessions=rows dataframe  
`sf$ sigCovs[[1]]`

### Vectors and single numbers

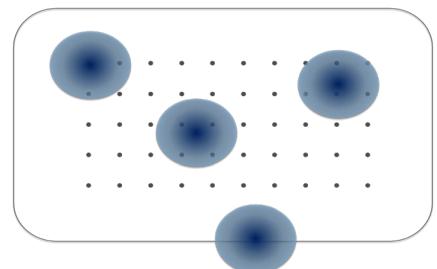
```
sf$ occasions
sf$mmdm
sf$mdm
```

## Datasets available

```
> data(package = "osCR")
> data(ocelot)
> data("beardata")
> data("nybears")
> data("peromyscus")
> data("mink")
```

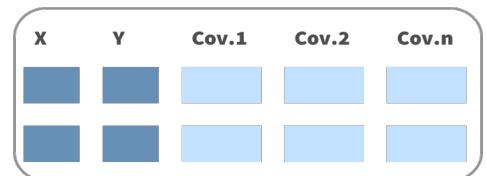
## 2. Create the State Space

The **State Space (S)** is the core element of SCR models. It defines where individuals can live and should represent activity centers of all sampled individuals.



### ssDF: the State Space Data Frame

- List with spatially explicit information from each session.
- At least include the coordinates (X, Y) of the discrete state space (UTM).
- Can include spatial covariates for a continuous state space to study variation in Density.
- Non habitat can be removed by removing unwanted coordinates (e.g., parking lot).

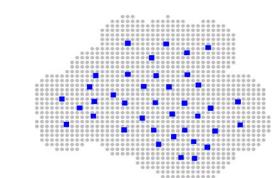


### 2.1. make.ssDF():

- Remember that  $\frac{1}{2} mmdm \sim \sigma$
  - Extracts covariates and removes non habitat
- ```
ss <- make.ssDF(scrFrame,
                  buffer, #~3 to 4σ around traps
                  res) # ≤ σ
```

2.2. Plot the state space

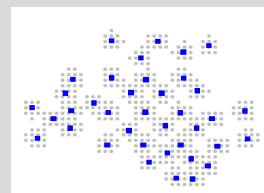
- Plot state space
`plot(ss)`
- Plot state space & traps
`plot(ss, sf)`



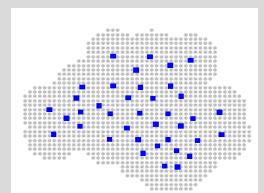
Vary the buffer and/or resolution

💡 Varying buffer, fixed resolution

```
make.ssDF(sf,
           buffer = 1,
           res = 0.5)
```

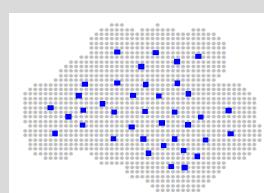


```
make.ssDF(sf,
           buffer = 3,
           res = 0.5)
```

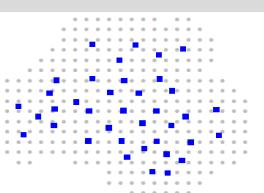


🔍 Fixed buffer, varying resolution

```
make.ssDF(sf,
           buffer = 3,
           res = 0.1)
```



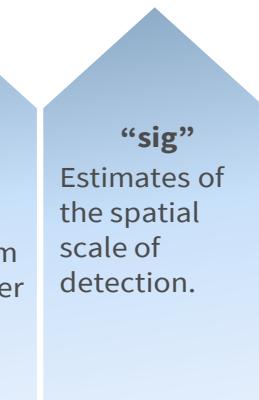
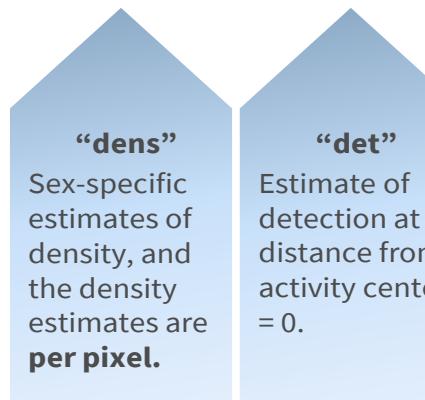
```
make.ssDF(sf,
           buffer = 3,
           res = 0.5)
```



3.2. Backtransform to the real scale

```
get.real(model,
          newdata,
          d.factor,
          type)
```

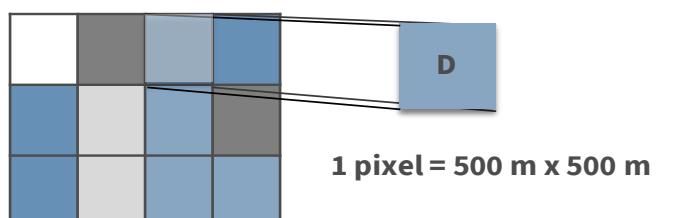
model	fitted model
newdata	Optional new data object for predictions
d.factor	optional scale the estimates to a different resolution
type	density ("dens"), detection probability ("det"), sigma("sig")



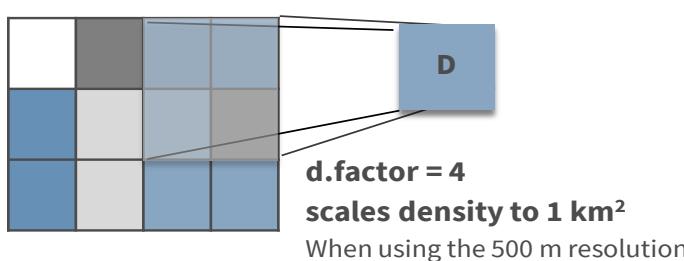
"sig"

Estimates of the spatial scale of detection.

d.factor



1 pixel = 500 m x 500 m





osCR :: CHEAT SHEET

Page 3 describes the specific functions and workflow for the null model and multi-session model in the osCR package.

Model specifics

Null model (SCR_0)

- The null model assumes homogeneous density which means all pixels have the same expected density.
- For additional arguments see `?osCR.fit()`

```
mod1 <- osCR.fit(list(D ~ 1,
p0 ~ 1, sig ~ 1),
scrFrame, #scrFrame object
ssDF, #ssDF object
... ) #other arguments
mod1 #summary
```



- If you included sex as a covariate in the scrFrame:
- Sex ratio psi() will be included in the summary
 - Can compare AIC with and without sex effects

Multi-session model

Are your data organized in multi-sessions and you want to analyze all of them jointly?



Spatial sessions: different study areas (e.g., parks, trapping grids)



Temporal sessions: same areas different times (e.g. seasons, years)



Session specific **population size** N_g (g=group/session)

- Test for differences among sessions using AIC.
- Can share parameters among sessions or not.

- The **multi-session** model follows similar steps as the single session model.
- The **edf** files from multiple sessions may be merged into one data frame prior to `data2oscr`
`edf <- rbind(edf1, edf2, ...)`
- The **tdf** files must be separate files for each session.

1. data2oscr for multi-session scrFrame

```
data <- data2oscr(
  edf, # include session column
  list(tdf1, tdf2, ...), # tdf files
  sess.col*, # session col in edf
  id.col*, # individual ID col in edf
  occ.col, # occasion col in edf
  trap.col*, # detector col in edf
  sex.col*, # sex col in edf
  sex.nacode, # unknown sex in edf
  K, # vector with occasions per session
  ntraps) # vector with traps per session
```

```
sf <- data$sf
sf # summary info per session (S1, S2..)
```

1.2. Summary of multi-session scrFrame

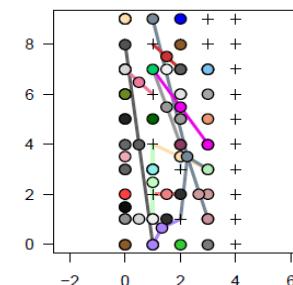
	S1	S2	S3	S4
n individuals	77	60	108	54
n traps	50	50	50	50
n occasions	7	5	6	4
avg caps	1.91	1.47	1.71	1.37
avg spatial caps	1.30	1.15	1.27	1.13
mmdm	2.57	2.32	1.76	2.84
Pooled MMDM:	2.21			

1.3. Plot spatial captures in a multi-session scrFrame

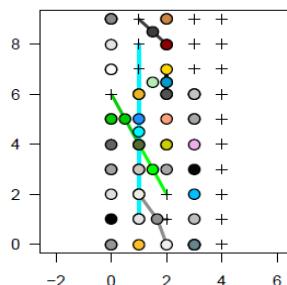
- Use `plot(sf)` to plot a spatial capture per session

```
par(mfrow=c(1,n)) # n = sessions
plot(sf) # plot all sessions
```

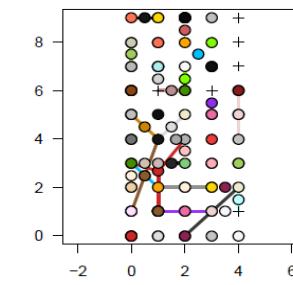
Session 1



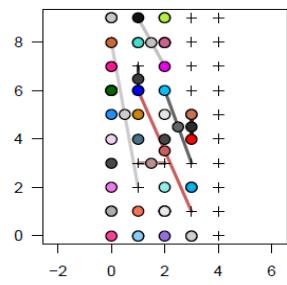
Session 2



Session 3



Session 4



2. Make the State Space Data Frame of a multi-session scrFrame

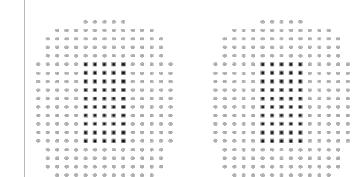
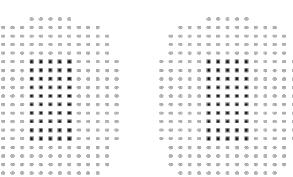
```
ss <- make.ssDF(
  scrFrame, # multi-session
  buffer, #buffer width
  res) #state space resolution
```

- You can vary the buffer and resolution as in the single-session model.

`?make.ssDF()` # Look at the help file for other arguments

- Visualize the state space

```
par(mfrow=c(1,n)) # n = sessions
plot.ssDF(ss, # state space
          sf) # traps
```



3. Model fitting

- Specify models that consider or not variation among sessions.

- fixed vs. session specific **D**
- fixed vs. session specific **p0**
- fixed vs. session specific **space use (σ)**

Model	Algebra	In <code>osCR.fit</code>
Density	$\log(D(s_i)) = \beta_0$	$D \sim 1$
Density	$\log(D(s_i)) = \beta_0 + \beta_{1(g)} Session_g$	$D \sim session$
Detection	$\text{logit}(p_0) = \alpha_0$	$p0 \sim 1$
Detection	$\text{logit}(p_0) = \alpha_0 + \alpha_{1(g)} Session_g$	$p0 \sim session$
Space use	$\log(\sigma) = \gamma_0$	$\text{sig} \sim 1$
Space use	$\log(\sigma) = \gamma_0 + \gamma_{1(g)} Session_g$	$\text{sig} \sim session$

- Include all models into a list using `fitList.osCR()`:

```
f1 <- fitList.osCR(
  mods, # list of fitted models
  rename) # if TRUE models are renamed with sensible names
```

- Compare multiple models
`ms <- modSel.osCR(f1)`

- Generate an AIC table to compare multiple models
`ms$aic`

- Generate a coefficient table
`ms$coef.tab`

- Generate a model averaged coefficients
`ma <- ma.coef(ms) # include a modSel.osCR object`

3.1. Back transform to the real scale

```
top.model <- m3
```

```
pred.df <- data.frame(session =
  factor (c(1, 2, 3, 4, ...)))
```

```
pred.det <- get.real(
  model = top.model, type = "det",
  newdata = pred.df)
```

Generate Tables

overview_tab generates a data frame that collapses the time condition for each id by taking into account potential gaps in the time frame

id	time	Var1	Var2
A	1990		
A	1991		
A	1992		
B	1990		

id	time
A	1990 - 1992
B	1990

```
output_table <-  
  overview_tab(  
    dat = toydata,  
    id = ccode,  
    time = year)
```

add data frame
define your time and scope variables

overview_crosstab generates a cross table that divides the data based on two conditions

id	time		
		Yes	No
Yes			
No			

```
output_crosstab <-  
  overview_crosstab(  
    dat = toydata,  
    cond1 = gdp,  
    cond2 = population,  
    threshold1 = 25000,  
    threshold2 = 27000,  
    id = ccode,  
    time = year  
)
```

define your conditions with cond1 and cond2
set your thresholds

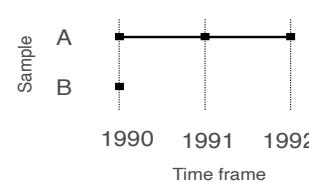
Note, if a data set is used that has multiple observations on the id-time unit, the function automatically aggregates the data set using the mean of condition 1 (**cond1**) and condition 2 (**cond2**).

If you store your results in an object, you can use **overview_print** to export them to a LaTeX output.

Generate Plots

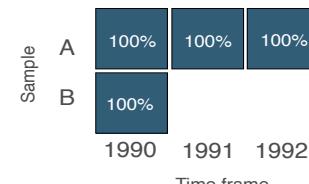
Sample overview

overview_plot illustrates the information that is generated in **overview_table** in a ggplot2 graphic



```
overview_plot(  
  dat = toydata,  
  id = ccode,  
  time = year)
```

overview_heat is similar to **overview_plot** but presents the frequency of data points by id-time-unit in a heat map



```
overview_heat(  
  dat = toydata,  
  id = ccode,  
  time = year,  
  perc = TRUE,  
  exp_total = 12)
```

displays percentage
max observations by id-time unit

Export Results

Tables

overview_print generates a LaTeX output (works with both **overview_tab** and **overview_crosstab** output)

```
overview_print(  
  obj = output_table)
```

```
overview_print(  
  obj = output_crosstab,  
  crosstab = TRUE)
```

TRUE for cross tables

The table can be modified with the **title**, **id**, **time**, **cond1**, and **cond2** arguments to replace default names

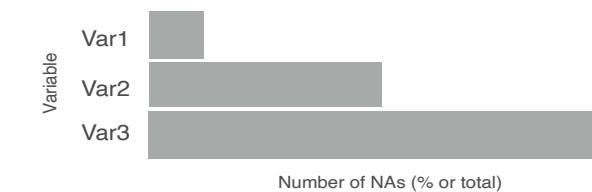
It also allows to save your output in a .tex file

```
overview_print(  
  obj = output_table,  
  save_out = TRUE,  
  path = "SET-YOUR-PATH",  
  file = "output.tex")
```

define where your output should be stored

Missing values (NAs)

overview_na returns a horizontal ggplot2 bar plot that indicates the amount of missing data (NAs) for each variable



Number of NAs (% or total)

relative distribution

```
overview_na(toydata_with_na)
```

```
overview_na(toydata_with_na,  
           perc = FALSE)
```

FALSE gives total number

The outputs of **overview_tab** and **overview_crosstab** are also compatible with other packages and functions such as **xtable**, **flextable**, or **kable** from **knitr**.

To generate a table in Rmarkdown with **knitr::kable**:

```
knitr::kable(output_table)
```

Plots

As the plots are based on ggplot2, plots can be stored with **ggplot2::ggsave**

```
ggplot2::ggsave(  
  output_plot,  
  filename = "FILENAME.png")
```

add plot object
add filename

Alternatively, storing the object also works this way:

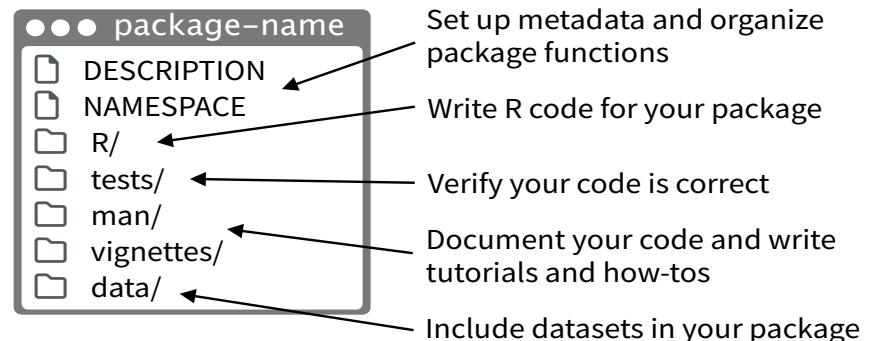
```
png("FILENAME.png")
```

```
output_plot
```

```
dev.off()
```

Package Structure

A package is a convention for organizing files into directories. This cheat sheet shows how to work with the 7 most common parts of an R package:



There are multiple packages useful to package development, including **usethis** which handily automates many of the more repetitive tasks. Load and install **devtools** which wraps together several of these packages to access everything in one step.

Getting Started

Once per machine:

- Get set up with **use_r_profile()**

```
if (interactive()) {
  require("devtools", quietly = TRUE)
  # automatically attaches usethis
}
```

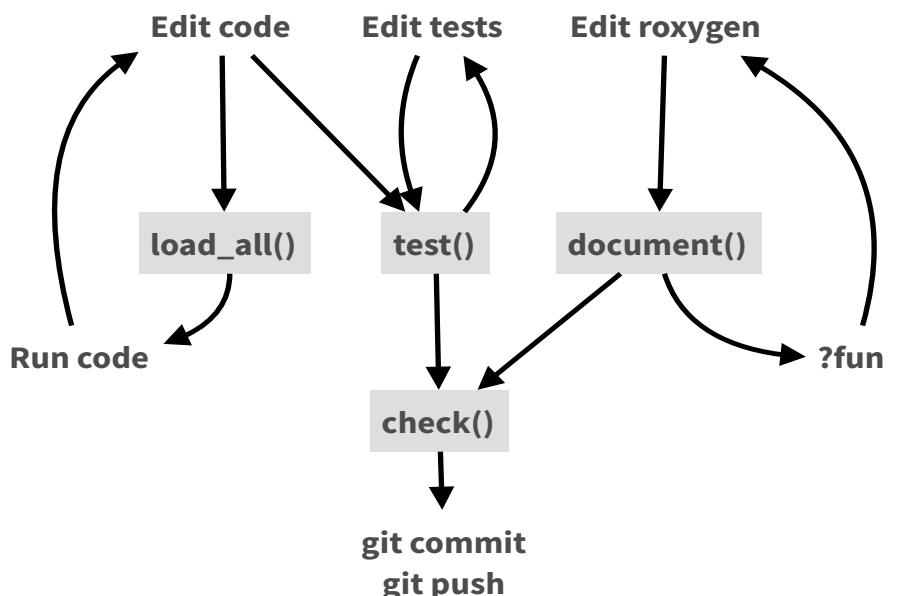
- create_github_token()** — Set up GitHub credentials
- git_vaccinate()** — Ignores common special files

Once per package:

- use_git()** — Activate git
- use_github()** — Connect to GitHub
- use_github_actions()** — Set up automated checks

Having problems with git? Get a situation report with **git_sitrep()**.

Workflow



- load_all()** (Ctrl/Cmd + Shift + L) — Load code
- document()** (Ctrl/Cmd + Shift + D) — Rebuild docs and NAMESPACE
- test()** (Ctrl/Cmd + Shift + T) — Run tests
- check()** (Ctrl/Cmd + Shift + E) — Check complete package

R/

All of the R code in your package goes in **R/**. A package with just an R/ directory is still a very useful package.

- Create a new package project with **create_package("path/to/name")**.
- Create R files with **use_r("file-name")**.

- Follow the tidyverse style guide at style.tidyverse.org
- Click on a function and press **F2** to open its definition
- Find a function or file with **Ctrl +**.

DESCRIPTION

The **DESCRIPTION** file describes your work, sets up how your package will work with other packages, and applies a license.

- Pick a license with **use_mit_license()**, **use_gpl3_license()**, **use_proprietary_license()**.
- Add packages that you need with **use_package()**.

Import packages users of your package need. R will install them when it installs your package.

Suggest packages that developers of your package need. Users can install or not, as they like.

NAMESPACE

The **NAMESPACE** file helps you make your package self-contained: it won't interfere with other packages, and other packages won't interfere with it.

- Export functions for users by placing **@export** in their roxygen comments.
- Access objects from other packages with **package::object** or **@importFrom package object** (recommended) or **@import package** (use with caution).
- Call **document()** to generate NAMESPACE and **load_all()** to reload.

DESCRIPTION

Makes **packages** available
Mandatory
use_package()

NAMESPACE

Makes **function** available
Optional (can use :: instead)
use_import_from()

The documentation will become the help pages in your package.

- Document each function with a roxygen block above its definition. In RStudio, Code > Insert Roxygen Skeleton helps.
- Document each dataset with roxygen block above the name of the dataset in quotes.
- Document the package with `use_package_doc()`.

vignettes/

- Create a vignette that is included with your package with `use_vignette()`.
- Create an article that only appears on the website with `use_article()`.
- Write the body of your vignettes in R Markdown.

Websites with pkgdown



- Use GitHub and `use_pkdown_github_pages()` to set up pkdown and configures an automated workflow using GitHub Actions and Pages.
- If you're not using GitHub, call `use_pkdown()` to configure pkdown. Then build locally with `pkdown::build_site()`.

tests/



- Create a test file with `use_test()`.
- Write tests with `test_that()` and `expect_()`.
- Run all tests with `test()` and run tests for current file with `test_active_file()`.
- See coverage of all files with `test_coverage()` and see coverage of current file with `test_coverage_active_file()`.

The **roxygen2** package lets you write documentation inline in your .R files with shorthand syntax.

- Add roxygen documentation as comments beginning with `#'`.
- Place a roxygen `@` tag (right) after `#'` to supply a specific section of documentation.
- Untagged paragraphs will be used to generate a title, description, and details section (in that order).

```
'#' Add together two numbers
'#'
'#' @param x A number.
'#' @param y A number.
'#' @returns The sum of `x` and `y`.
'#' @export
'#' @examples
'#' add(1, 1)
add <- function(x, y) {
  x + y
}
```

COMMON ROXYGEN TAGS

<code>@description</code>	<code>@family</code>	<code>@returns</code>
<code>@examples</code>	<code>@inheritParams</code>	<code>@seealso</code>
<code>@examplesIf</code>	<code>@param</code>	
<code>@export</code>	<code>@rdname</code>	

README.Rmd + NEWS.md

- Create a README and NEWS markdown files with `use_readme_rmd()` and `use_news_md()`.

Expect statement

`expect_equal()`
`expect_error()`
`expect_snapshot()`

Tests

Is equal? (within numerical tolerance)
Throws specified error?
Output is unchanged?

```
test_that("Math works", {
  expect_equal(1 + 1, 2)
  expect_equal(1 + 2, 3)
  expect_equal(1 + 3, 4)
})
```

- Record how a data set was prepared as an R script and save that script to `data-raw/` with `use_data_raw()`.
- Save a prepared data object to `data/` with `use_data()`.

Package States

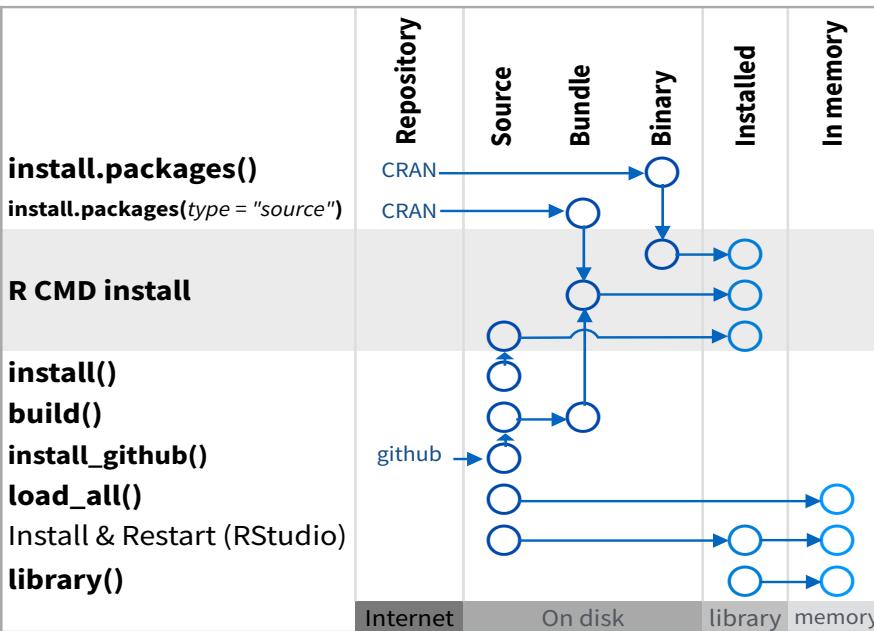
The contents of a package can be stored on disk as a:

- **source** - a directory with sub-directories (as shown in Package structure)
- **bundle** - a single compressed file (`.tar.gz`)
- **binary** - a single compressed file optimized for a specific OS

Packages exist in those states locally or remotely, e.g. on CRAN or on GitHub.

From those states, a package can be installed into an R library and then loaded into memory during an R session.

Use the functions below to move between these states.



Visit r-pkgs.org to learn much more about writing and publishing packages for R.

Searching CRAN with packagefinder: : CHEAT SHEET



CONSOLE

```
findPackage(keywords, mode = "or", case.sensitive = FALSE, always.sensitive = NULL,
weights = c(2,2,1,2), display = "viewer", results.longdesc = FALSE, limit.results = 15, silent =
FALSE, index = NULL, advanced.ranking = TRUE, return.df = FALSE, clipboard = FALSE)
```

Most important arguments

keywords Word or vector of words to search for

mode Find packages with every keyword ("and") or with any of the keywords ("or")?
Will be overruled if keywords contain logical operators like keywords = "X and Y"

case.sensitive Case-sensitive search?

always.sensitive Vector of words that will always be treated as case-sensitive, e.g. abbreviations

limit.results How many results to display in console or viewer?

Outputs

display

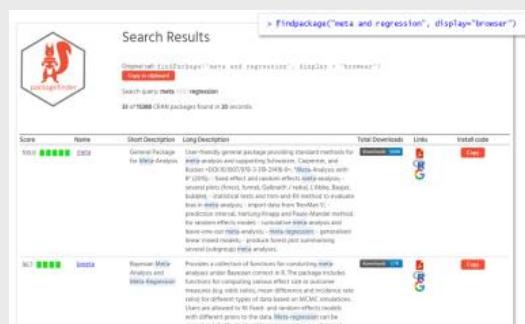
display = "console"

Score	Name	Short Description	GO
100	xml2	Parse XML	16203
93	XML	Tools for Parsing and Generating XML Within R and S-Plus	16356
92.9	XML	Tools for Parsing and Generating XML Within R and S-Plus	16202
77.6	xmrlr	Read, Write and work with "XML" Data	16207
66	XML2R	Easier XML data collection	16204
65.7	xmlrpc2	Implementation of the Remote Procedure Call Protocol ("XML-RPC")	16208
54.8	flatxml	Tools for working with XML Files as R Dataframes	4639

display = "viewer"

Score	Name	Short Description	GO
100	xml2	Parse XML	16203
93	XML	Tools for Parsing and Generating XML Within R and S-Plus	16356
92.9	XML	Tools for Parsing and Generating XML Within R and S-Plus	16202
77.6	xmrlr	Read, Write and Work with "XML" Data	16207
66	XML2R	Easier XML data collection	16204
65.7	xmlrpc2	Implementation of the Remote Procedure Call Protocol (XML-RPC)	16208
54.8	flatxml	Tools for working with XML Files as R Dataframes	4639

display = "browser"



return.df = TRUE Return results as dataframe

clipboard = TRUE Copy results to clipboard

Examples

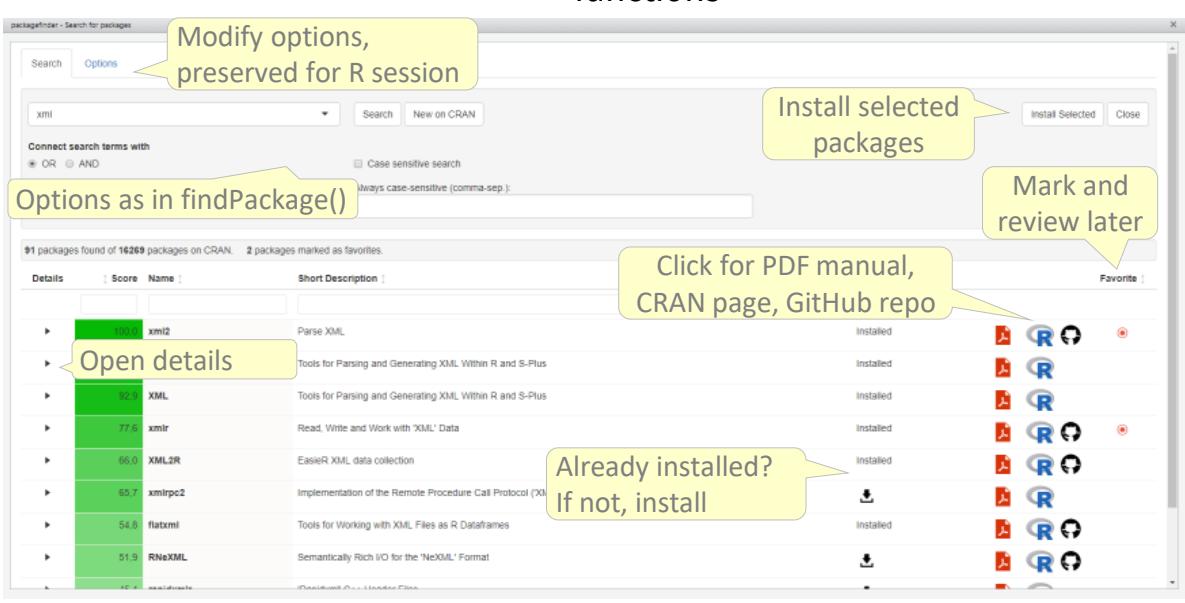
- > findPackage("parameters", mode = "and", always.sensitive = "SEM")
- > findPackage("meta and regression")
- > my.results <- findPackage(c("meta", "regression"), "and", return.df = TRUE)
- > findPackage("xml", display = "browser")

RSTUDIO ADD-IN



Automatically installed with the package.

Provides a graphical interface to the findPackage() and whatsNew() functions



ADDITIONAL FUNCTIONS

whatsNew(last.days = 0) Show new packages on CRAN

packageDetails(package) Show details of a CRAN package in the console

lastResults(package = "viewer") Show results of last search again

fp(...) Short hand for findPackage(...)

go(package, where.to = "details") Install CRAN package, show PDF manual, details or package website

Parallel Computing :: CHEAT SHEET

Splitting :

Splitting a code by :

1. **Task** (different tasks on same data)
2. **Data** (one task on different data)

Hardware needs :

CPU (+2 cores)

RAM (shared memory vs distributed memory)

2 ideas in parallel computing :

1. Map-Reduced Models :

(distributed data; physically on different devices)

- Hadoop
- Spark

R Packages:

- sparklyr, iotools
- pbdr (programming with big data in R)

2. Master - Worker Models :

(M tasks on C cores; usually $1 < C \ll M$)

R Packages:

- snow, snowFT, snowfall
- foreach
- future, future.apply



Not always parallel computing:

stop/start cluster takes time

overhead (communication time b/w master and workers ; not good for repeatedly sending big data!)

Sequential vs Parallel:

```
library(microbenchmark)
microbenchmark( FUN1(...), FUN2(...),
times = 10)
```

parallel.R : core package

```
library(parallel)
ncores <- detectCores(logical=F) # physical cores
cl <- makeCluster(ncores)
clusterApply(cl, x = c(...), fun = FUN) # FUN(x,...)
stopCluster(cl)
```

Initialization of workers :

```
clusterCall(cl,FUN) # calls FUN on workers
clusterEvalQ(cl, exp) # eval an exp. on workers
## clusterEvalQ(cl, library(foo))
clusterExport(cl, varlist) # varlist on workers
## clusterExport(cl, c("mean")) where mean = 10
```

Data Chunk on workers :

1. generated on workers


```
# clusterApply(cl,x, FUN) e.g FUN(){ rnorm()}
```
2. generated on master and pass to workers


```
# ind <- splitIndices(200, 5)
# clusterApply(cl, ind, FUN)
# (-) : not efficient in Big Data : heavy
```
3. chunk on workers # copy of original Data on all workers


```
# clusterExport(cl, M)      e.g. M is a matrix
# clusterApply(cl, x, FUN)   FUN contains subset M
```

foreach.R : Sequential

```
library(foreach)      # by default return a list
foreach(n = rep(5,3), m = 10^(0:2)) %do% FUN(n,m)
foreach(n, .packages = "X") %do% FUN(n)
# FUN needs package X to be run
foreach(n, .export = c("Y")) %do% FUN(n,b=Y)
# FUN needs outside object/function "Y"
foreach(n,.combine = rbind) %do% FUN(n) #row bind
foreach(n,.combine = '+') %do% FUN(n) #rbind + colSum
foreach(n,.combine = c) %do% FUN(n) # vector
foreach(n,.combine = c) %::% when(n > 2) %do% FUN(n)
```

future.R : asynchronously

```
library(future) (variables run as soon as created)
plan(multicore)
# plans : sequential, cluster, multicore, multiprocess
x %<-% mean(rnorm(100))
y %<-% mean(rnorm(100))
```

future.apply.R : parallel_apply

```
library(future.apply) (parallel _apply functions)
plan(multicore) # can be other plans
future_apply(n,FUN),future_lapply(...),future_sapply(...)
```

foreach.R : Parallel

needs backend packages support parallel computing

- doParallel(parallel.R), doFuture (future.R), doSEQ

doParallel.R : backend of foreach

```
library(doParallel)
cl <- makeCluster(ncores)      # ncores = 2,3,..
registerDoParallel(cl)        # register the backend
foreach(...) %dopar% FUN(...)
```

doFuture.R : backend of foreach

```
library(doFuture)
registerDoFuture()
plan(cluster , workers = 3) # can be other plans
foreach(...) %dopar% FUN(...)
```

Load Balancing: for uneven task times

```
clusterApplyLB(cl,x,FUN) # not for small task time
clusterApply(cl, x = splitIndices(10,2), FUN)
library(itertools)
foreach(s=isplitVector(1:10, chunks = 2))%dopar% FUN
# e.g. FUN = sapply(s,"*",100)
future_sapply(..., future.scheduling = 1)
```

Introduction to REST APIs

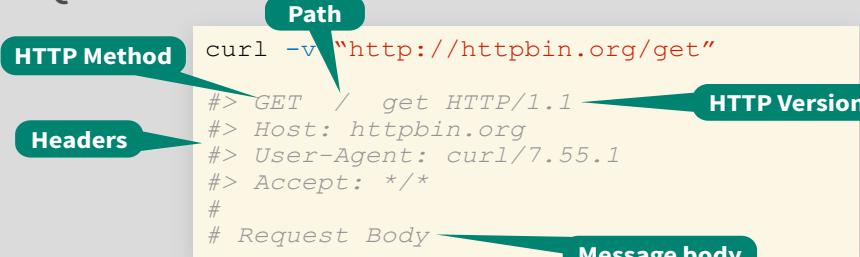
Web APIs use **HTTP** to communicate between **client** and **server**.

HTTP



HTTP is built around a **request** and a **response**. A **client** makes a request to a **server**, which handles the request and provides a response. Requests and responses are specially formatted text containing details and data about the exchange between client and server.

REQUEST



RESPONSE



Plumber: Build APIs with R

Plumber uses special comments to turn any arbitrary R code into API endpoints. The example below defines a function that takes the `msg` argument and returns it embedded in additional text.

```
Plumber comments begin with #*
library(plumber)
#* @apiTitle Plumber Example API
#* Echo back the input
#* @param msg The message to echo
#* @get /echo
function(msg = "") {
  list(
    msg = paste0(
      "The message is: '", msg, "'"))
}
```

R Studio

Plumber pipeline

Plumber endpoints contain R code that is executed in response to an HTTP request. Incoming requests pass through a set of mechanisms before a response is returned to the client.

FILTERS

Filters can forward requests (after potentially mutating them), throw errors, or return a response without forwarding the request. Filters are defined similarly to endpoints using the `@filter [name]` tag. By default, filters apply to all endpoints. Endpoints can opt out of filters using the `@preempt` tag.

PARSER

Parsers determine how Plumber parses the incoming request body. By default Plumber parses the request body as JavaScript Object Notation (JSON). Other parsers, including custom parsers, are identified using the `@parser [parser name]` tag. All registered parsers can be viewed with `registered_parsers()`.

ENDPOINT

Endpoints define the R code that is executed in response to incoming requests. These endpoints correspond to HTTP methods and respond to incoming requests that match the defined method.

METHODS

- `@get` - request a resource
- `@post` - send data in body
- `@put` - store / update data
- `@delete` - delete resource
- `@head` - no request body
- `@options` - describe options
- `@patch` - partial changes
- `@use` - use all methods

SERIALIZER

Serializers determine how Plumber returns results to the client. By default Plumber serializes the R object returned into JavaScript Object Notation (JSON). Other serializers, including custom serializers, are identified using the `@serializer [serializer name]` tag. All registered serializers can be viewed with `registered_serializers()`.

Identify as filter

Forward request

Endpoint description

Parser

HTTP Method

Filter name

```
library(plumber)
#* @filter log
function(req, res) {
  print(req$HTTP_USER_AGENT)
  forward()
}

#* Convert request body to uppercase
#* @preempt log
#* @parser json
#* @post /uppercase
#* @serializer json
function(req, res) {
  toupper(req$body)
}
```

Opt out of the log filter

Endpoint path

Serializer

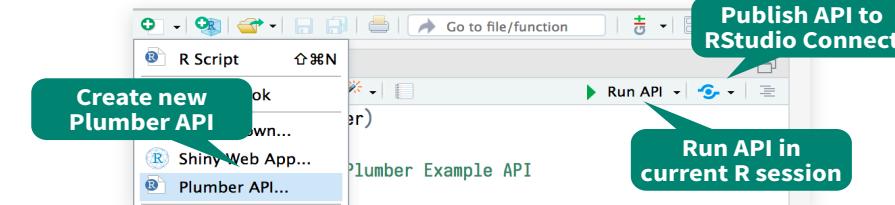
Running Plumber APIs

Plumber APIs can be run programmatically from within an R session.

```
library(plumber)
plumb("plumber.R") %>%
  pr_run(port = 5762)
```

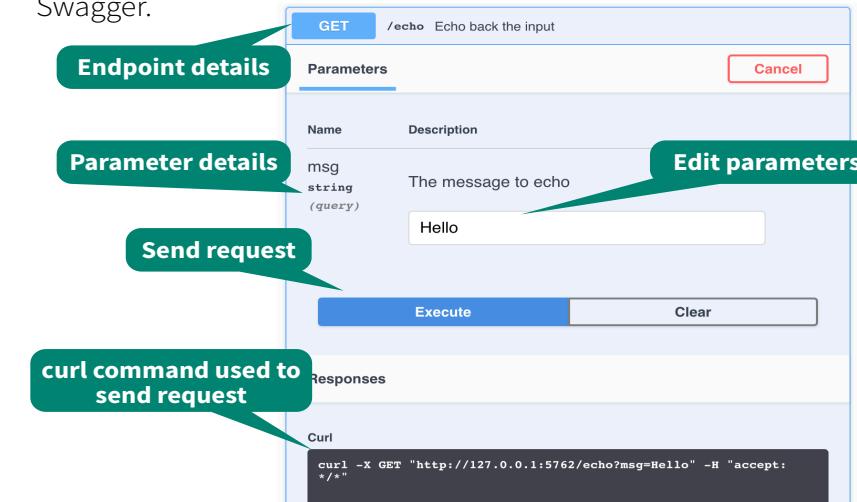
This runs the API on the host machine supported by the current R session.

IDE INTEGRATION



Documentation

Plumber APIs automatically generate an OpenAPI specification file. This specification file can be interpreted to generate a dynamic user-interface for the API. The default interface is generated via Swagger.



Interact with the API

Once the API is running, it can be interacted with using any HTTP client. Note that using `httr` requires using a separate R session from the one serving the API.

```
(resp <- httr:::GET("localhost:5762/echo?msg=Hello"))
#> Response [http://localhost:5762/echo?msg=Hello]
#>   Date: 2018-08-07 20:06
#>   Status: 200
#>   Content-Type: application/json
#>   Size: 35 B
httr:::content(resp, as = "text")
#> [1] "{\\"msg\\":\\"The message is: 'Hello'\\\"}"
```



Plumber is exceptionally customizable. In addition to using special comments to create APIs, APIs can be created entirely programmatically. This exposes additional features and functionality. Plumber has a convenient “tidy” interface that allows API routers to be built piece by piece. The following example is part of a standard `plumber.R` file.

```
library(plumber)

#* @plumber
function(pr) {
  pr %>%
    pr_get(path = "/echo",
           handler = function(msg = "") {
             list(msg = paste0(
               "The message is: '", msg, "'"))
           })
  pr %>%
    pr_get(path = "/plot",
           handler = function() {
             rand <- rnorm(100)
             hist(rand)
           },
           serializer = serializer_png()) %>%
    pr_post(path = "/sum",
            handler = function(a, b) {
              as.numeric(a) + as.numeric(b)
            })
}


```

OpenAPI

Plumber automatically creates an OpenAPI specification file based on Plumber comments. This file can be further modified using `pr_set_api_spec()` with either a function that modifies the existing specification or a path to a `.yaml` or `.json` specification file.

```
library(plumber)

#* @param msg The message to echo
#* @get /echo
function(msg = "") {
  list(
    msg = paste0(
      "The message is: '", msg, "'"))
}

#* @plumber
function(pr) {
  pr %>%
    pr_set_api_spec(function(spec) {
      spec$paths[["/echo"]る$get$summary <-
        "Echo back the input"
      spec})
}


```

By default, Swagger is used to interpret the OpenAPI specification file and generate the user interface for the API. Other interpreters can be used to adjust the look and feel of the user interface via `pr_set_docs()`.



Plumber provides access to special `req` and `res` objects that can be passed to Plumber functions. These objects provide access to the request submitted by the client and the response that will be sent to the client. Each object has several components, the most helpful of which are outlined below:

Name	Example	Description
req		
req\$pr	plumber::pr()	The Plumber router processing the request
req\$body	list(a=1)	Typically the same as <code>argsBody</code>
req\$argsBody	list(a=1)	The parsed body output
req\$argsPath	list(c=3)	The values of the path arguments
req\$argsQuery	list(e=5)	The parsed output from <code>req\$QUERY_STRING</code>
req\$cookies	list(cook = "a")	A list of cookies
req\$REQUEST_METHOD	"GET"	The method used for the HTTP request
req\$PATH_INFO	"/"	The path of the incoming HTTP request
req\$HTTP_*	"HTTP_USER_AGENT"	All of the HTTP headers sent with the request
req\$bodyRaw	charToRaw("a=1")	The <code>raw()</code> contents of the request body
res		
res\$headers	list(header = "abc")	HTTP headers to include in the response
res\$setHeader()	setHeader("foo", "bar")	Sets an HTTP header
res\$setCookie()	setCookie("foo", "bar")	Sets an HTTP cookie on the client
res\$removeCookie	removeCookie("foo")	Removes an HTTP cookie
res\$body	"{\\"a\\": [1]}"	Serialized output
res\$status	200	The response HTTP status code
res\$toResponse()	toResponse()	A list of status, headers, and body

Plumber supports asynchronous execution via the `future` R package. This pattern allows Plumber to concurrently process multiple requests.

```
library(plumber)
future::plan("multisession")

#* @get /slow
function() {
  promises::future_promise({
    slow_calc()
  })
}
```

Set the execution plan

Slow calculation

MOUNTING ROUTERS

Plumber routers can be combined by mounting routers into other routers. This can be beneficial when building routers that involve several different endpoints and you want to break each component out into a separate router. These separate routers can even be separate files loaded using `plumb()`.

```
library(plumber)
route <- pr() %>%
  pr_get("/foo", function() "foo")

#* @plumber
function(pr) {
  pr %>%
    pr_mount("/bar", route)
}
```

Create an initial router

Mount one router into another

In the above example, the final route is `/bar/foo`.

RUNNING EXAMPLES

Some packages, like the Plumber package itself, may include example Plumber APIs. Available APIs can be viewed using `available_apis()`. These example APIs can be run with `plumb_api()` combined with `pr_run()`.

```
library(plumber)
plumb_api(package = "plumber",
          name = "01-append",
          edit = TRUE) %>%
  pr_run()
```

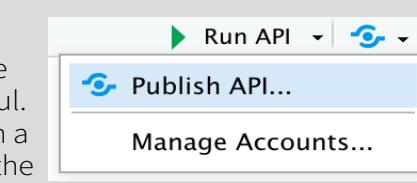
Identify the package name and API name

Run the example API

Optionally open the file for editing

Deploying Plumber APIs

Once Plumber APIs have been developed, they often need to be deployed somewhere to be useful. Plumber APIs can be deployed in a variety of different ways. One of the easiest way to deploy Plumber APIs is using RStudio Connect, which supports push button publishing from the RStudio IDE.





Map Functions

ONE LIST

map(.x, .f, ...) Apply a function to each element of a list or vector, return a list.
`x <- list(1:10, 11:20, 21:30)
l1 <- list(x = c("a", "b"), y = c("c", "d"))
map(l1, sort, decreasing = TRUE)`



map_dbl(.x, .f, ...)
Return a double vector.
`map_dbl(x, mean)`

map_int(.x, .f, ...)
Return an integer vector.
`map_int(x, length)`

map_chr(.x, .f, ...)
Return a character vector.
`map_chr(l1, paste, collapse = "")`

map_lgl(.x, .f, ...)
Return a logical vector.
`map_lgl(x, is.integer)`

map_dfc(.x, .f, ...)
Return a data frame created by column-binding.
`map_dfc(l1, rep, 3)`

map_dfr(.x, .f, ..., .id = NULL)
Return a data frame created by row-binding.
`map_dfr(x, summary)`

walk(.x, .f, ...) Trigger side effects, return invisibly.
`walk(x, print)`

Function Shortcuts

Use `~.` with functions like **map()** that have single arguments.

`map(l, ~. + 2)`
becomes
`map(l, function(x) x + 2)`



R Studio

Use `~.x .y` with functions like **map2()** that have two arguments.

`map2(l, p, ~.x +.y)`
becomes
`map2(l, p, function(l, p) l + p)`

Use a **string** or an **integer** with any map function to index list elements by name or position. **map(l, "name")** becomes `map(l, function(x) x[["name"]])`

TWO LISTS

map2(.x, .y, .f, ...) Apply a function to pairs of elements from two lists or vectors, return a list.
`y <- list(1, 2, 3); z <- list(4, 5, 6); l2 <- list(x = "a", y = "z")
map2(x, y, ~.x * .y)`



map2_dbl(.x, .y, .f, ...)
Return a double vector.
`map2_dbl(y, z, ~.x / .y)`

map2_int(.x, .y, .f, ...)
Return an integer vector.
`map2_int(y, z, `+`)`

map2_chr(.x, .y, .f, ...)
Return a character vector.
`map2_chr(l1, l2, paste, collapse = "", sep = ":".)`

map2_lgl(.x, .y, .f, ...)
Return a logical vector.
`map2_lgl(l2, l1, `%in%`)`

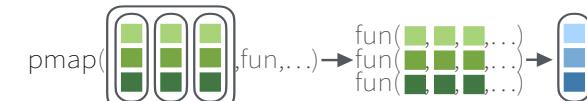
map2_dfc(.x, .y, .f, ...)
Return a data frame created by column-binding.
`map2_dfc(l1, l2, ~ as.data.frame(c(x, y)))`

map2_dfr(.x, .y, .f, ..., .id = NULL)
Return a data frame created by row-binding.
`map2_dfr(l1, l2, ~ as.data.frame(c(x, y)))`

walk2(.x, .y, .f, ...) Trigger side effects, return invisibly.
`walk2(objs, paths, save)`

MANY LISTS

pmap(.l, .f, ...) Apply a function to groups of elements from a list of lists or vectors, return a list.
`pmap(list(x, y, z), ~ ..1 * (.2 + ..3))`



pmap_dbl(.l, .f, ...)
Return a double vector.
`pmap_dbl(list(y, z), ~.x / .y)`

pmap_int(.l, .f, ...)
Return an integer vector.
`pmap_int(list(y, z), `+`)`

pmap_chr(.l, .f, ...)
Return a character vector.
`pmap_chr(list(l1, l2), paste, collapse = "", sep = ":".)`

pmap_lgl(.l, .f, ...)
Return a logical vector.
`pmap_lgl(list(l2, l1), `%in%`)`

pmap_dfc(.l, .f, ...) Return a data frame created by column-binding.
`pmap_dfc(list(l1, l2), ~ as.data.frame(c(x, y)))`

pmap_dfr(.l, .f, ..., .id = NULL) Return a data frame created by row-binding.
`pmap_dfr(list(l1, l2), ~ as.data.frame(c(x, y)))`

pwalk(.l, .f, ...) Trigger side effects, return invisibly.
`pwalk(list(objs, paths), save)`

LISTS AND INDEXES

imap(.x, .f, ...) Apply `.f` to each element and its index, return a list.
`imap(y, ~ paste0(y, ": ", .x))`



imap_dbl(.x, .f, ...)
Return a double vector.
`imap_dbl(y, ~ .y)`

imap_int(.x, .f, ...)
Return an integer vector.
`imap_int(y, ~ .y)`

imap_chr(.x, .f, ...)
Return a character vector.
`imap_chr(y, ~ paste0(y, ": ", .x))`

imap_lgl(.x, .f, ...)
Return a logical vector.
`imap_lgl(l1, ~ is.character(.y))`

imap_dfc(.x, .f, ...)
Return a data frame created by column-binding.
`imap_dfc(l2, ~ as.data.frame(c(x, y)))`

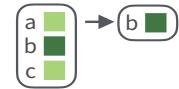
imap_dfr(.x, .f, ..., .id = NULL)
Return a data frame created by row-binding.
`imap_dfr(l2, ~ as.data.frame(c(x, y)))`

iwalk(.x, .f, ...) Trigger side effects, return invisibly.
`iwalk(z, ~ print(paste0(y, ": ", .x)))`

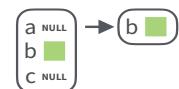
Use `~.x .y` with functions like **imap()**. `.x` will get the list value and `.y` will get the index, or name if available.

imap(list(a, b, c), ~ paste0(.y, ": ", .x))
outputs "index: value" for each item

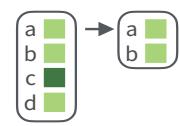
Filter



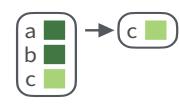
keep(.x, .p, ...)
Select elements that pass a logical test.
Conversely, **discard()**.
`keep(x, is.na)`



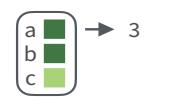
compact(.x, .p = identity)
Drop empty elements.
`compact(x)`



head_while(.x, .p, ...)
Return head elements until one does not pass.
Also **tail_while()**.
`head_while(x, is.character)`



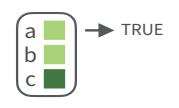
detect(.x, .f, ..., dir = c("forward", "backward"), .right = NULL, .default = NULL)
Find first element to pass.
`detect(x, is.character)`



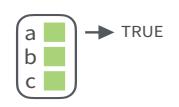
detect_index(.x, .f, ..., dir = c("forward", "backward"), .right = NULL)
Find index of first element to pass.
`detect_index(x, is.character)`



every(.x, .p, ...)
Do all elements pass a test?
`every(x, is.character)`



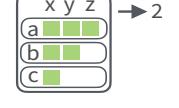
some(.x, .p, ...)
Do some elements pass a test?
`some(x, is.character)`



none(.x, .p, ...)
Do no elements pass a test?
`none(x, is.character)`

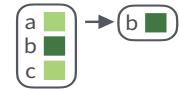


has_element(.x, .y)
Does a list contain an element?
`has_element(x, "foo")`

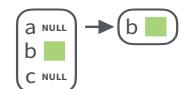


vec_depth(x)
Return depth (number of levels of indexes).
`vec_depth(x)`

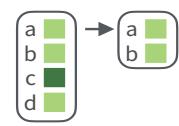
Index



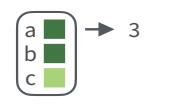
pluck(.x, ..., .default=NULL)
Select an element by name or index. Also **attr_getter()** and **chuck()**.
`pluck(x, "b")`
`x %>% pluck("b")`



assign_in(x, where, value)
Assign a value to a location using pluck selection.
`assign_in(x, "b", 5)`
`x %>% assign_in("b", 5)`



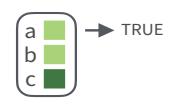
modify_in(.x, where, f)
Apply a function to a value at a selected location.
`modify_in(x, "b", abs)`
`x %>% modify_in("b", abs)`



flatten(.x) Remove a level of indexes from a list.
Also **flatten_chr()** etc.
`flatten(x)`



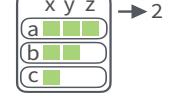
array_tree(array, margin = NULL) Turn array into list.
Also **array_branch()**.
`array_tree(x, margin = 3)`



cross2(.x, .y, .filter = NULL)
All combinations of .x and .y.
Also **cross()**, **cross3()**, and **cross_df()**.
`cross2(1:3, 4:6)`

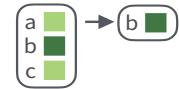


transpose(.l, .names = NULL)
Transposes the index order in a multi-level list.
`transpose(x)`

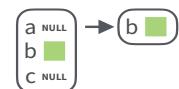


set_names(x, nm = x)
Set the names of a vector/list directly or with a function.
`set_names(x, c("p", "q", "r"))`
`set_names(x, tolower)`

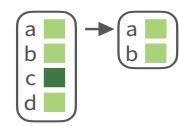
Modify



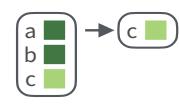
modify(.x, .f, ...) Apply a function to each element. Also **modify2()**, and **imodify()**.
`modify(x, ~.+ 2)`



modify_at(.x, .at, .f, ...) Apply a function to selected elements.
Also **map_at()**.
`modify_at(x, "b", ~.+ 2)`

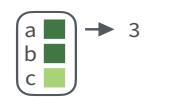


modify_if(.x, .p, .f, ...) Apply a function to elements that pass a test. Also **map_if()**.
`modify_if(x, is.numeric, ~.+2)`



modify_depth(.x, .depth, .f, ...) Apply function to each element at a given level of a list. Also **map_depth()**.
`modify_depth(x, 2, ~.+ 2)`

Reshape



append(x, values, after = length(x)) Add values to end of list.
`append(x, list(d = 1))`



prepend(x, values, before = 1) Add values to start of list.
`prepend(x, list(d = 1))`

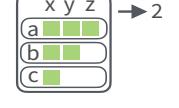
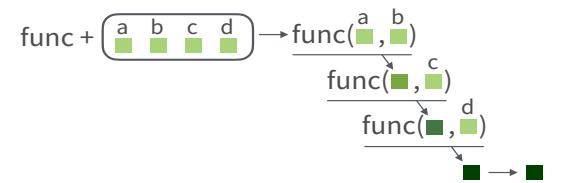


splice(...) Combine objects into a list, storing S3 objects as sub-lists.
`splice(x, y, "foo")`

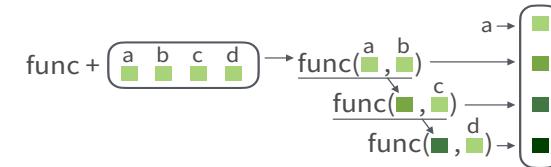


Reduce

reduce(.x, .f, ..., .init, .dir = c("forward", "backward")) Apply function recursively to each element of a list or vector. Also **reduce2()**.
`reduce(x, sum)`



accumulate(.x, .f, ..., .init) Reduce a list, but also return intermediate results. Also **accumulate2()**.
`accumulate(x, sum)`



List-Columns



List-columns are columns of a data frame where each element is a list or vector instead of an atomic value. Columns can also be lists of data frames. See **tidyverse** for more about nested data and list columns.

WORK WITH LIST-COLUMNS

Manipulate list-columns like any other kind of column, using **dplyr** functions like **mutate()** and **transmute()**. Because each element is a list, use **map functions** within a column function to manipulate each element.

map(), **map2()**, or **pmap()** return lists and will create new list-columns.

list function,
return list

starwars %>%
transmute(ships = map2(vehicles,
starships,
append))

column function

list function,
return int

starwars %>%
mutate(n_films = map_int(films, length))

column function

list-column

quanteda Cheat Sheet

General syntax

- **corpus_*** manage text collections/metadata
 - **tokens_*** create/modify tokenized texts
 - **dfm_*** create/modify doc-feature matrices
 - **fcm_*** work with co-occurrence matrices
 - **textstat_*** calculate text-based statistics
 - **textmodel_*** fit (un-)supervised models
 - **textplot_*** create text-based visualizations
- Consistent grammar:**
- **object()** constructor for the object type
 - **object_verb()** inputs & returns object type

Extensions

- quanteda** works well with these companion packages:
- **quanteda.textmodels**: Text scaling and classification models
 - **readtext**: an easy way to read text data
 - **spacyr**: NLP using the spaCy library
 - **quanteda.corpora**: additional text corpora
 - **stopwords**: multilingual stopword lists in R

Create a corpus from texts (corpus_*)

Read texts (txt, pdf, csv, doc, docx, json, xml)

```
my_texts <- readtext::readtext("~/link/to/path/*")
```

Construct a corpus from a character vector

```
x <- corpus(data_char_ukimmig2010, text_field = "text")
```

Explore a corpus

```
summary(data_corpus_inaugural, n = 2)
## Corpus consisting of 58 documents, showing 2 documents:
##
##           Text Types Tokens Sentences Year President FirstName Party
## 1789-Washington 625    1537      23 1789 Washington George   none
## 1793-Washington  96     147       4 1793 Washington George   none
```

Extract or add document-level variables

```
party <- data_corpus_inaugural$Party
x$serial_number <- seq_len(ndoc(x))
docvars(x, "serial_number") <- seq_len(ndoc(x)) # alternative
```

Bind or subset corpora

```
corpus(x[1:5]) + corpus(x[7:9])
corpus_subset(x, Year > 1990)
```

Change units of a corpus

```
corpus_reshape(x, to = "sentences")
```

Segment texts on a pattern match

```
corpus_segment(x, pattern, valuetype, extract_pattern = TRUE)
```

Take a random sample of corpus texts

```
corpus_sample(x, size = 10, replace = FALSE)
```

Extract features (dfm_*; fcm_*)

Create a document-feature matrix (dfm) from a corpus

```
x <- dfm(data_corpus_inaugural,
tolower = TRUE, stem = FALSE, remove_punct = TRUE,
remove = stopwords("english"))
```

```
print(x, max_ndoc = 2, max_nfeat = 4)
## Document-feature matrix of: 58 documents, 9,210 features (92.6% sparse) and 4 docvars.
##               features
## docs          fellow-citizens senate house representatives
## 1789-Washington      1     1     2      2
## 1793-Washington      0     0     0      0
## [ reached max_ndoc ... 56 more documents, reached max_nfeat ... 9,206 more features ]
```

Create a dictionary

```
dictionary(list(negative = c("bad", "awful", "sad"),
positive = c("good", "wonderful", "happy")))
```

Apply a dictionary

```
dfm_lookup(x, dictionary = data_dictionary_LSD2015)
```

Select features

```
dfm_select(x, pattern = data_dictionary_LSD2015, selection = "keep")
```

Randomly sample documents or features

```
dfm_sample(x, what = c("documents", "features"))
```

Weight or smooth the feature frequencies

```
dfm_weight(x, scheme = "prop") | dfm_smooth(x, smoothing = 0.5)
```

Sort or group a dfm

```
dfm_sort(x, margin = c("features", "documents", "both"))
dfm_group(x, groups = "President")
```

Combine identical dimension elements of a dfm

```
dfm_compress(x, margin = c("both", "documents", "features"))
```

Create a feature co-occurrence matrix (fcm)

```
x <- fcm(data_corpus_inaugural, context = "window", size = 5)
fcm_compress/fcm_norm/fcm_select/fcm_upper/fcm_lower are also available
```

Useful additional functions

Locate keywords-in-context

```
kwic(data_corpus_inaugural, pattern = "america*")
```

Utility functions

texts(corpus)	Show texts of a corpus
ndoc(corpus / dfm / tokens)	Count documents/features
nfeat(corpus / dfm / tokens)	Count features
summary(corpus / dfm)	Print summary
head(corpus / dfm)	Return first part
tail(corpus / dfm)	Return last part



PGS Catalog access with quincunx

Introduction

The **PGS Catalog** is a service provided by the EMBL-EBI and University of Cambridge that offers a manually curated and freely available database of published polygenic scores (PGS): <https://www.pgscatalog.org/>.

The PGS Catalog data provided by the **REST API** is organised around five core entities:

- PGS** Polygenic Scores
- PGP** PGS Publications
- PSS** PGS Sample Sets
- PPM** PGS Performance Metrics
- EFO** EFO traits

Get PGS Catalog Entities

quincunx facilitates the access to the Catalog via the REST API, allowing you to programmatically retrieve data directly into R. Each of the five entities is mapped to an S4 object of a class of the same name.



Query criteria for retrieval functions, e.g., PGS can be queried by either pgs_id, efo_id or pubmed_id. These correspond to the criteria exposed by the PGS Catalog REST API: <https://www.pgscatalog.org/rest/>.

Search by	Example	PGS	PGP	PSS	PPM	EFO
pgs_id	"PGS000001"	■	■	■	■	■
pgp_id	"PGP000001"		■			
pss_id	"PSS000001"			■		
ppm_id	"PPM000001"				■	
efo_id	"EFO_0000249"	■				■
pubmed_id	"25855707"	■	■			
author	"Mavaddat"		■			
trait_term	"Alzheimer"					■

PGS Catalog Entities in R

PGS Catalog entities are represented as S4 classes in R. Each class represents a relational database of tidy data tables. All objects start with a table with the same name as the class. Combination of variables indicated in bold renders each row unique in each table.

S4 class scores

scores	samples	demographics
• pgs_id	• pgs_id	• pgs_id
• pgs_name	• sample_id	• sample_id
• scoring_file	• stage	• variable
• matches_publication	• sample_size	• estimate_type
• reported_trait	• sample_cases	• estimate
• trait_additional_description	• sample_controls	• unit
• pgs_method_name	• sample_percent_male	• variability_type
• pgs_method_params	• phenotype_description	• variability
• n_variants	• ancestry_category	• interval_type
• n_variants_interactions	• ancestry	• interval_lower
• assembly	• country	• interval_upper
• license	• ancestry_additional_description	
• beta_unit	• study_id	
	• pubmed_id	
	• cohorts_additional_description	
publications		cohorts
• pgs_id		• pgs_id
• pgp_id		• sample_id
• pubmed_id		• cohort_symbol
• publication_date		• cohort_name
• publication		
• title		
• author_fullname		
• doi		
traits		
• pgs_id		
• efo_id		
• trait		
• description		
• url		

S4 class publications

publications	pgs_ids
• pgp_id	• ppg_id
• pubmed_id	• pgs_id
• publication_date	• stage
• publication	
• title	
• author_fullname	
• doi	
• authors	

S4 class traits

traits	pgs_ids	child_pgs_ids
• efo_id	• efo_id	• efo_id
• parent_efo_id	• parent_efo_id	• parent_efo_id
• is_child	• is_child	• is_child
• trait	• trait	• trait
• description	• pgs_id	• child_pgs_id
• url		
3x trait_{categories, synonyms, mapped_terms}		
• efo_id		
• parent_efo_id		
• is_child		
• trait_{category, synonyms, mapped_terms}		

S4 class sample_sets

sample_sets	samples	demographics
• pss_id	• pss_id	• pss_id
• pgs_name	• sample_id	• sample_id
• scoring_file	• stage	• variable
• matches_publication	• sample_size	• estimate_type
• reported_trait	• sample_cases	• estimate
• trait_additional_description	• sample_controls	• unit
• pgs_method_name	• sample_percent_male	• variability_type
• pgs_method_params	• phenotype_description	• variability
• n_variants	• ancestry_category	• interval_type
• n_variants_interactions	• ancestry	• interval_lower
• assembly	• country	• interval_upper
• license	• ancestry_additional_description	
• beta_unit	• study_id	
	• pubmed_id	
	• cohorts_additional_description	
cohorts		
• pss_id		
• sample_id		
• cohort_symbol		
• cohort_name		

S4 class performance_metrics

performance_metrics	samples	demographics
• ppm_id	• ppm_id	• ppm_id
• pgs_id	• pss_id	• pss_id
• reported_trait	• sample_id	• sample_id
• covariates	• stage	• variable
• comments	• sample_size	• estimate_type
publications		
• ppm_id		
• pgp_id		
• pubmed_id		
• publication_date		
• publication		
• title		
• author_fullname		
• doi		
sample_sets		
• ppm_id		
• pss_id		
• sample_id		
• cohort_symbol		
• cohort_name		
3x pgs_{effect_sizes,classification_metrics,other_metrics}		
• ppm_id		
{effect_size_id,classification_metrics_id,other_metrics_id}		
• estimate_type_long		
• estimate_type		
• estimate		
• unit		
• variability_type		
• variability		



Other S4 Entities

Besides the five PGS Catalog entities, there are three other objects that can be retrieved from the REST API: trait_categories, cohorts and releases.

S4 class trait_categories

	trait_categories
•	trait_category
•	efo_id
•	trait
•	description
•	url

S4 class cohorts

	cohorts
•	cohort_symbol
•	cohort_name

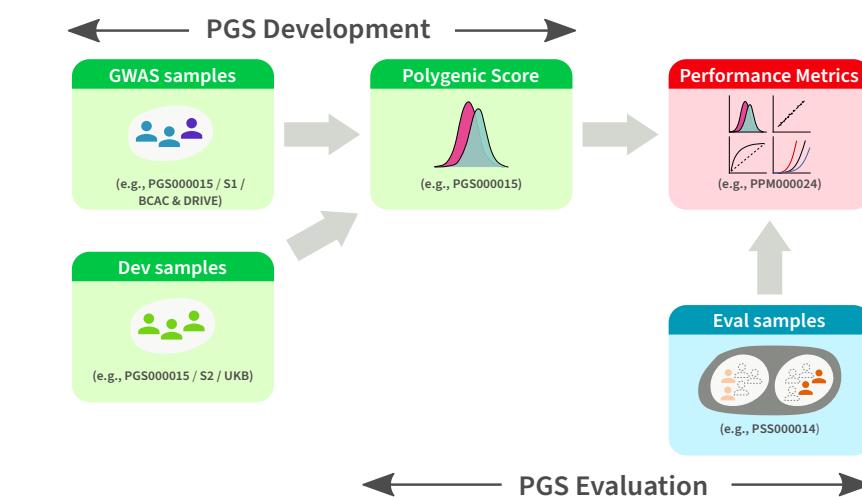
S4 class releases

	releases
•	date
•	n_pgs
•	n_ppm
•	n_ppg
•	notes

3x {pgs_ids, ppm_ids, ppg_ids}

- date
- {pgs_id, ppm_id, ppg_id}

PGS Construction Process



Samples and Polygenic Scores (PGS) are annotated according to their utilisation context in the PGS construction process, i.e. the stage variable in quincunx:

- Source of Variant Associations (GWAS): stage="gwas"
- Score Development/Training: stage="dev"
- Development: stage="gwas/dev" ("gwas" and "dev")
- PGS Evaluation: stage="eval"

Cohorts, Samples and Sample Sets

Cohorts

A cohort is a group of individuals with a shared characteristic. Cohorts are identified in quincunx by the cohort_symbol variable.



Samples

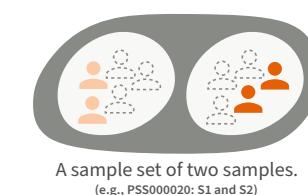
A sample is a group of participants associated with none, one or more catalogued cohorts. The selection from a cohort can be either a subset or its totality. Samples are not identified in PGS Catalog with a global unique identifier, but quincunx assigns a surrogate identifier (sample_id) to allow relations between tables.

Possible compositions of samples:



Sample Sets

A sample set is a group of samples used in a polygenic score evaluation. Each sample set is identified in the PGS Catalog by a unique sample set identifier (PSS ID).



Manipulate Cases of S4 Entities

Get a scores object s consisting of two polygenic scores (PGS):

```
s <- get_scores(pgs_id = c('a', 'b'))
```

Subset object s by either identifier or position using '[':

	→	
<code>s['a'] # Subset by identifier</code>		<code>s[1] # Subset by position</code>

Combine two scores' objects:

	+		→	
			(Discards duplicates)	
			(Keeps duplicates)	<code>bind(s1, s2)</code>

Polygenic scoring file

PGS scoring files are provided by the PGS Catalog to allow computation of polygenic scores by users. These files are hosted at the PGS Catalog FTP server: <http://ftp.ebi.ac.uk/pub/databases/spot/pgs/scores/>. They are labelled by their respective PGS Score ID (e.g. PGS000001.txt.gz). For more details please visit: <https://maialab.org/quincunx/articles/pgs-scoring-file.html>.

File Format

Each scoring file contains variant identification, effect alleles and respective score weights. The file is formatted as a gzipped tab-delimited text file, with a header containing brief metadata about the score. You can read PGS scoring files into R with `read_scoring_file()`.

PGS000117.txt.gz

```

1  ### PGS CATALOG SCORING FILE - see www.pgscatalog.org/downloads/#dl_ftp for...
2  ## POLYGENIC SCORE (PGS) INFORMATION
3  # PGS ID = PGS000117
4  # Reported Trait = Cardiovascular Disease
5  # Original Genome Build = GRCh37
6  # Number of Variants = 267863
7  ## SOURCE INFORMATION
8  # PGP ID = PGP000054
9  # Citation = Elliott J et al. JAMA (2020). doi:10.1001/jama.2019.22241
10 rsID      chr_name chr_position effect_allele reference_allele effect_weight
11 rs11240779 1       808631   A       G       0.00077622
12 rs1921     1       949608   A       G       -0.00583829
13 rs2710890  1       958905   G       A       -0.00182583
14 rs4970349  1       967658   T       C       -0.001855691
...

```

Columns

The following table lists all possible columns in a PGS scoring file. A few columns are required (R), and most are optional (O); either the rsID alone or the combination of chr_name and chr_position are required, with the other being optional.

Column (Requirement)	Description	Example
rsID (R/O)	dbSNP Accession ID	"rs554219"
chr_name (R/O)	Chromosome name	"11"
chr_position (R/O)	Chromosome position	69516874
effect_allele (R)	Effect allele	"G"
reference_allele (O)	Reference allele	"C"
effect_weight (R)	Variant weight	0.117
locus_name (O)	Locus name	"CCND1"
weight_type (O)	Type of weight	"log(OR)", "beta_cox"
allelefrequency_effect (O)	Effect allele frequency	0.410
is_interaction (O)	Variant interaction?	TRUE or FALSE
is_recessive (O)	Recessive inheritance model?	TRUE or FALSE
is_haplotype (O)	Is effect allele a haplotype?	TRUE or FALSE
is_diploid (O)	Is effect allele a diploid?	TRUE or FALSE
imputation_method (O)	Imputation method	TODO
variant_description (O)	Variant description	TODO
inclusion_criteria (O)	Score inclusion criteria	TODO
OR (O)	Odds Ratio	1.12
HR (O)	Hazard Ratio	1.08

Two Arm Trials

Simple random assignment is like flipping coins for each unit separately.

```
simple_ra(N = 100, prob = 0.5)
```

Complete random assignment allocates a fixed number of units to each condition.

```
complete_ra(N = 100, m = 50)
complete_ra(N = 100, prob = 0.5)
```

Block random assignment conducts complete random assignment separately for groups of units.

```
blocks <- rep(c("A", "B", "C"),
              c(50, 100, 200))

# defaults to half of each block
block_ra(blocks = blocks)

# can change with block_m
block_ra(blocks = blocks,
         block_m = c(20, 30, 40))
```

Cluster random assignment allocates whole groups of units to conditions together.

```
clusters <- rep(letters, times = 1:26
cluster_ra(clusters = clusters)
```

Block and cluster random assignment conducts cluster random assignment separately for groups of clusters.

```
clusters <- rep(letters, times = 1:26)
blocks <- rep(paste0("block_", 1:5),
             c(15, 40, 65, 90, 141))
block_and_cluster_ra(blocks = blocks,
                     clusters = clusters)
```

randomizr is part of the DeclareDesign suite of packages for designing, implementing, and analyzing social science research designs.

Multi Arm Trials

Set the number of arms with `num_arms` or with `conditions`.

```
complete_ra(N = 100, num_arms = 3)
complete_ra(N = 100, conditions = c("control",
                                    "placebo", "treatment"))
```

The `*_each` arguments in `randomizr` functions specify design parameters for each arm separately.

```
complete_ra(N = 100, m_each = c(20, 30, 50))
complete_ra(N = 100,
           prob_each = c(0.2, 0.3, 0.5))
```

If the design is the **same** for all blocks, use `prob_each`:

```
blocks <- rep(c("A", "B", "C"),
              c(50, 100, 200))
block_ra(blocks = blocks,
         prob_each = c(.1, .1, .8))
```

If the design is **different** in different blocks, use `block_m_each` or `block_prob_each`:

```
block_m_each <- rbind(c(10, 20, 20),
                      c(30, 50, 20),
                      c(50, 75, 75))
block_ra(blocks = blocks,
         block_m_each = block_m_each)

block_prob_each <- rbind(c(.1, .1, .8),
                         c(.2, .2, .6),
                         c(.3, .3, .4))
block_ra(blocks = blocks,
         block_prob_each = block_prob_each)
```

If `conditions` is numeric, the output will be **numeric**.

If `conditions` is not numeric, the output will be a **factor** with levels in the order provided to `conditions`.

```
complete_ra(N = 100, conditions = -2:2)
complete_ra(N = 100, conditions = c("A", "B"))
```

Declaration

Learn about assignment procedures by “declaring” them with `declare_ra()`

```
declaration <-
  declare_ra(N = 100, m_each = c(30, 30, 40))
```

```
declaration # print design information
```

Conduct a random assignment:

```
conduct_ra(declaration)
```

Obtain observed condition probabilities (useful for inverse probability weighting if probabilities of assignment are not constant)

```
Z <- conduct_ra(declaration)
obtain_condition_probabilities(declaration, Z)
```

Sampling

All assignment functions have sampling analogues: Sampling is identical to a two arm trial where the treatment group is sampled.

Assignment

```
simple_ra()
complete_ra()
block_ra()
cluster_ra()
block_and_cluster_ra()
declare_ra()
conduct_ra()
```

Sampling

```
simple_rs()
complete_rs()
strata_rs()
cluster_rs()
strata_and_cluster_rs()
declare_rs()
draw_rs()
```

Stata

A Stata version of `randomizr` is available, with the same arguments but different syntax:

```
ssc install randomizr
set obs 100
complete_ra, m(50)
```

Detect pattern



string

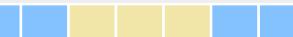
Locate pattern



Extract pattern



Replace pattern



```
> string <- c("Hipopotamus", "Rhymoceros", "time for bottomless lyrics")
> pattern <- "t.m"
```

Detect Patterns

grep(pattern, string)

[1] 1 3

grep(pattern, string, value = TRUE)

```
[1] "Hipopotamus"
[2] "time for bottomless lyrics"
```

grepl(pattern, string)

[1] TRUE FALSE TRUE

stringr::str_detect(string, pattern)

[1] TRUE FALSE TRUE

Split a String using a Pattern

strsplit(string, pattern) or stringr::str_split(string, pattern)

Locate Patterns

regexpr(pattern, string)

find starting position and length of first match

gregexpr(pattern, string)

find starting position and length of all matches

stringr::str_locate(string, pattern)

find starting and end position of first match

stringr::str_locate_all(string, pattern)

find starting and end position of all matches

regmatches(string, regexpr(pattern, string))

extract first match

[1] "tam" "tim"

regmatches(string, gregexpr(pattern, string))

extract all matches, outputs a list

[[1]] "tam" [[2]] character(0) [[3]] "tim" "tom"

stringr::str_extract(string, pattern)

extract first match

[1] "tam" NA "tim"

stringr::str_extract_all(string, pattern)

extract all matches, outputs a list

stringr::str_extract_all(string, pattern, simplify = TRUE)

extract all matches, outputs a matrix

stringr::str_match(string, pattern)

extract first match + individual character groups

stringr::str_match_all(string, pattern)

extract all matches + individual character groups

Replace Patterns

sub(pattern, replacement, string)

replace first match

gsub(pattern, replacement, string)

replace all matches

stringr::str_replace(string, pattern, replacement)

replace first match

stringr::str_replace_all(string, pattern, replacement)

replace all matches

Special Metacharacters

\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed

Lookarounds and Conditionals*

(?=)	Lookahead (requires PERL = TRUE), e.g. (?=yx): position followed by 'xy'
(?!)	Negative lookahead (PERL = TRUE); position NOT followed by pattern
(?<=)	Lookbehind (PERL = TRUE), e.g. (?<=yx): position following 'xy'
(?<!=)	Negative lookbehind (PERL = TRUE); position NOT following pattern
?(if)then	If-then-condition (PERL = TRUE); use lookaheads, optional char. etc in if-clause
?(if)then else	If-then-else-condition (PERL = TRUE)

*see, e.g. <http://www.regular-expressions.info/lookaround.html>
<http://www.regular-expressions.info/conditional.html>

Character Classes and Groups

.	Any character except \n
	Or, e.g. (a b)
[...]	List permitted characters, e.g. [abc]
[a-z]	Specify character ranges
[^...]	List excluded characters
(...)	Grouping, enables back referencing using \\N where N is an integer

General Modes

By default R uses *extended regular expressions*. You can switch to *PCRE regular expressions* using PERL = TRUE for base or by wrapping patterns with perl() for stringr.

All functions can be used with literal searches using fixed = TRUE for base or by wrapping patterns with fixed() for stringr.

All base functions can be made case insensitive by specifying ignore.case = TRUE.

Anchors

^	Start of the string
\$	End of the string
\b	Empty string at either edge of a word
\B	NOT the edge of a word
\B	Beginning of a word
\B	End of a word

Escaping Characters

Metacharacters (. * + etc.) can be used as literal characters by escaping them. Characters can be escaped using \\ or by enclosing them in \\Q...\\E.

Case Conversions

Regular expressions can be made case insensitive using (?i). In backreferences, the strings can be converted to lower or upper case using \\L or \\U (e.g. \\L\\1). This requires PERL = TRUE.

Quantifiers

*	Matches at least 0 times
+	Matches at least 1 time
?	Matches at most 1 time; optional string
{n}	Matches exactly n times
{n,}	Matches at least n times
{n,m}	Matches between n and m times

Greedy Matching

By default the asterisk * is greedy, i.e. it always matches the longest possible string. It can be used in lazy mode by adding ?, i.e. *?.

Greedy mode can be turned off using (?U). This switches the syntax, so that (?U)a* is lazy and (?U)a*? is greedy.

Note

Regular expressions can conveniently be created using e.g. the packages rex or rebus.

The **reticulate** package lets you use Python and R together seamlessly in R code, in R Markdown documents, and in the RStudio IDE.

Python in R Markdown

(Optional) Build Python env to use.

Add `knitr::knit_engines$set(python = reticulate::eng_python)` to the setup chunk to set up the reticulate Python engine (not required for `knitr >= 1.18`).

Suggest the Python environment to use, in your setup chunk.

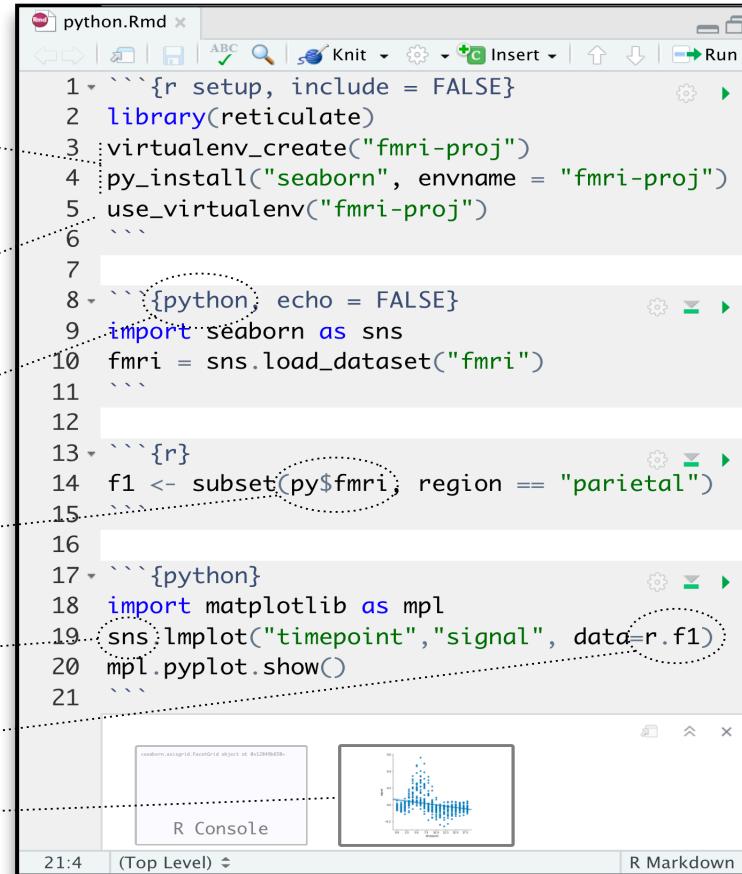
Begin Python chunks with ````{python}`. Chunk options like `echo`, `include`, etc. all work as expected.

Use the `py` object to access objects created in Python chunks from R chunks.

Python chunks all execute within a **single** Python session so you have access to all objects created in previous chunks.

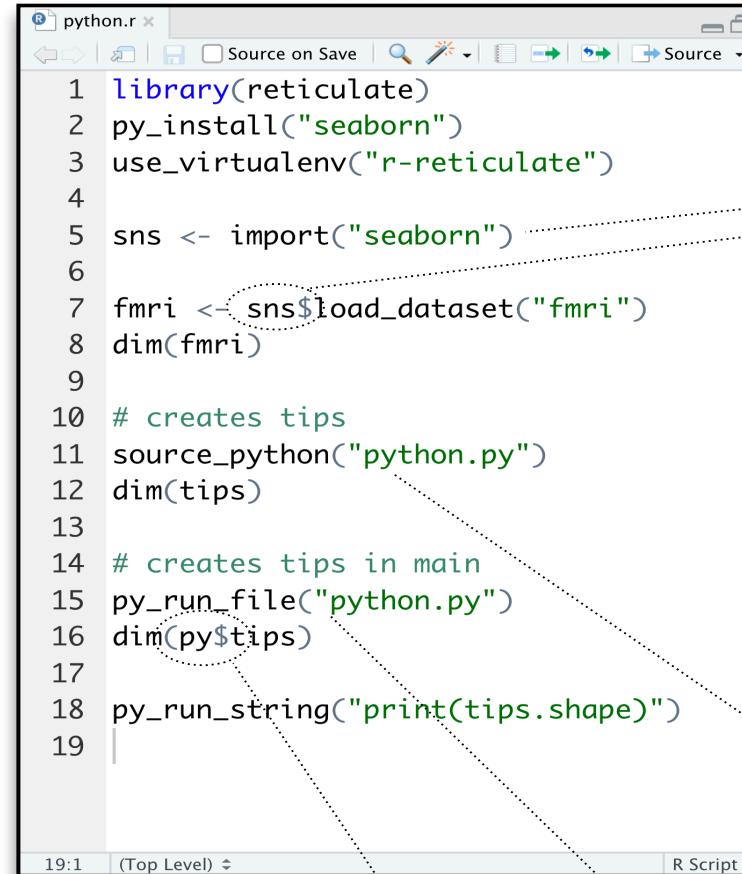
Use the `r` object to access objects created in R chunks from Python chunks.

Output displays below chunk, including matplotlib plots.



```
python.Rmd x
1  ```{r setup, include = FALSE}
2  library(reticulate)
3  virtualenv_create("fmri-proj")
4  py_install("seaborn", envname = "fmri-proj")
5  use_virtualenv("fmri-proj")
6  ...
7
8  ```{python echo = FALSE}
9  import seaborn as sns
10 fmri = sns.load_dataset("fmri")
11 ...
12
13 ```{r}
14 f1 <- subset(py$fmri, region == "parietal")
15 ...
16
17 ```{python}
18 import matplotlib as mpl
19 sns.lmplot("timepoint","signal", data=r.f1)
20 plt.show()
21 ```

R Console: A scatter plot of signal vs timepoint for the parietal region.
R Markdown: A histogram of signal values.
```



```
python.r x
1 library(reticulate)
2 py_install("seaborn")
3 use_virtualenv("r-reticulate")
4
5 sns <- import("seaborn")
6
7 fmri <- sns$load_dataset("fmri")
8 dim(fmri)
9
10 # creates tips
11 source_python("python.py")
12 dim(tips)
13
14 # creates tips in main
15 py_run_file("python.py")
16 dim(py$tips)
17
18 py_run_string("print(tips.shape)")
19

R Script: [Output pane showing the execution results of the R script code.]
```

Object Conversion

Tip: To index Python objects begin at 0, use integers, e.g. `0L`

Reticulate provides **automatic** built-in conversion between Python and R for many Python types.

R	↔	Python
Single-element vector		Scalar
Multi-element vector		List
List of multiple types		Tuple
Named list		Dict
Matrix/Array		NumPy ndarray
Data Frame		Pandas DataFrame
Function		Python function
NULL, TRUE, FALSE		None, True, False

Or, if you like, you can convert manually with

`py_to_r(x)` Convert a Python object to an R object. Also `r_to_py()`. `py_to_r(x)`

`tuple(..., convert = FALSE)` Create a Python tuple. `tuple("a", "b", "c")`

`dict(..., convert = FALSE)` Create a Python dictionary object. Also `py_dict()` to make a dictionary that uses Python objects as keys. `dict(foo = "bar", index = 42L)`

`np_array(data, dtype = NULL, order = "C")` Create NumPy arrays. `np_array(c(1:8), dtype = "float16")`

`array_reshape(x, dim, order = c("C", "F"))` Reshape a Python array. `x <- 1:4; array_reshape(x, c(2, 2))`

`py_func(f)` Wrap an R function in a Python function with the same signature. `py_func(xor)`

`py_main_thread_func(f)` Create a function that will always be called on the main thread.

`iterate(it, f = base::identity, simplify = TRUE)` Apply an R function to each value of a Python iterator or return the values as an R vector, draining the iterator as you go. Also `iter_next()` and `as_iterator()`. `iterate(iter, print)`

`py_iterator(fn, completed = NULL)` Create a Python iterator from an R function. `seq_gen <- function(x){ n <- x; function() {n <- n + 1; n}}; py_iterator(seq_gen(9))`

Helpers

`py_capture_output(expr, type = c("stdout", "stderr"))` Capture and return Python output. Also `py_suppress_warnings()`. `py_capture_output("x")`

`py_get_attr(x, name, silent = FALSE)` Get an attribute of a Python object. Also `py_set_attr()`, `py_has_attr()`, and `py_list_attributes()`. `py_get_attr(x)`

`py_help(object)` Open the documentation page for a Python object. `py_help(sns)`

`py_last_error()` Get the last Python error encountered. Also `py_clear_last_error()` to clear the last error. `py_last_error()`

`py_save_object(object, filename, pickle = "pickle", ...)` Save and load Python objects with pickle. Also `py_load_object()`. `py_save_object(x, "x.pickle")`

`with(data, expr, as = NULL, ...)` Evaluate an expression within a Python context manager. `py <- import_builtin(); with(py$open("output.txt", "w") %as% file, {file$write("Hello, there!")})`

Python in R

Call Python from R code in three ways:

IMPORT PYTHON MODULES

Use `import()` to import any Python module. Access the attributes of a module with `$`.

- `import(module, as = NULL, convert = TRUE, delay_load = FALSE)` Import a Python module. If `convert = TRUE`, Python objects are converted to their equivalent R types. Also `import_from_path()`. `import("pandas")`
- `import_main(convert = TRUE)` Import the main module, where Python executes code by default. `import_main()`
- `import_builtins(convert = TRUE)` Import Python's built-in functions. `import_builtins()`

SOURCE PYTHON FILES

Use `source_python()` to source a Python script and make the Python functions and objects it creates available in the calling R environment.

- `source_python(file, envir = parent.frame(), convert = TRUE)` Run a Python script, assigning objects to a specified R environment. `source_python("file.py")`

RUN PYTHON CODE

Execute Python code into the **main** Python module with `py_run_file()` or `py_run_string()`.

- `py_run_string(code, local = FALSE, convert = TRUE)` Run Python code (passed as a string) in the main module. `py_run_string("x = 10"); py$x`
- `py_run_file(file, local = FALSE, convert = TRUE)` Run Python file in the main module. `py_run_file("script.py")`
- `py_eval(code, convert = TRUE)` Run a Python expression, return the result. Also `py_call()`. `py_eval("1 + 1")`

Access the results, and anything else in Python's **main** module, with `py$`

- `py` An R object that contains the Python main module and the results stored there. `py$`

RStudio in the IDE Requires reticulate plus RStudio v1.2+. Some features require v1.4+.

Syntax highlighting for Python scripts and chunks.	Tab completion for Python functions and objects (and Python modules imported in R scripts).	Source Python scripts.	Execute Python code line by line with Cmd + Enter (Ctrl + Enter).	View Python objects in the Environment Pane.	View Python objects in the Data Viewer.
--	---	------------------------	---	--	---

A Python REPL opens in the console when you run Python code with a keyboard shortcut. Type **exit** to close.

matplotlib plots display in plots pane. Press **F1** over a Python symbol to display the help topic for that symbol.

Python REPL

A REPL (Read, Eval, Print Loop) is a command line where you can run Python code and view the results.

- Open in the console with **repl_python()**, or by running code in a Python script with **Cmd + Enter** (**Ctrl + Enter**).
- Type commands at **>>>** prompt.
- Press **Enter** to run code.
- Type **exit** to close and return to R console.

Configure Python

Reticulate binds to a local instance of Python when you first call **import()** directly or implicitly from an R session. To control the process, find or build your desired Python instance. Then suggest your instance to reticulate. **Restart R to unbind**.

Find Python

- install_python(version, list = FALSE, force = FALSE)** Download and install Python.
`install_python("3.6.13")`
- py_available(initialize = FALSE)** Check if Python is available on your system. Also **py_module_available()** and **py_numpy_module()**.
`py_available()`
- py_discover_config()** Return all detected versions of Python. Use **py_config()** to check which version has been loaded.
`py_config()`
- virtualenv_list()** List all available virtualenvs. Also **virtualenv_root()**.
`virtualenv_list()`
- conda_list(conda = "auto")** List all available conda envs. Also **conda_binary()** and **conda_version()**.
`conda_list()`

Create a Python env

- virtualenv_create(envname = NULL, ...)** Create a new virtual environment.
`virtualenv_create("r-pandas")`
- conda_create(envname = NULL, ...)** Create a new conda environment.
`conda_create("r-pandas", packages = "pandas")`

Install Packages

Install Python packages with R (below) or the shell:

```
pip install SciPy
conda install SciPy
```

- py_install(packages, envname, ...)** Installs Python packages into a Python env.
`py_install("pandas")`
- virtualenv_install(envname, packages, ...)** Install a package within a virtualenv. Also **virtualenv_remove()**.
`virtualenv_install("r-pandas", packages = "pandas")`
- conda_install(envname, packages, ...)** Install a package within a conda env. Also **conda_remove()**.
`conda_install("r-pandas", packages = "plotly")`

Suggest an env to use

Set a default Python interpreter in the RStudio IDE Global or Project Options.

Go to **Tools > Global Options... > Python** for Global Options.

Within a project, go to **Tools > Project Options... > Python**.

Otherwise, to choose an instance of Python to bind to, reticulate scans the instances on your computer in the following order, **stopping at the first instance that contains the module called by import()**.

- The instance referenced by the environment variable **RETICULATE PYTHON** (if specified). **Tip: set in .Renviron file.**
 - Sys.setenv(RETICULATE PYTHON = PATH)** Set default Python binary. Persists across sessions! Undo with **Sys.unsetenv()**.
`Sys.setenv(RETICULATE PYTHON = "/usr/local/bin/python")`
- The instances referenced by **use_** functions if called before **import()**. Will fail silently if called after **import** unless **required = TRUE**.
 - use_python(python, required = FALSE)** Suggest a Python binary to use by path.
`use_python("/usr/local/bin/python")`
 - use_virtualenv(virtualenv = NULL, required = FALSE)** Suggest a Python virtualenv.
`use_virtualenv("~/myenv")`
 - use_condaenv(condaenv = NULL, conda = "auto", required = FALSE)** Suggest a conda env to use. `use_condaenv(condaenv = "r-nlp", conda = "/opt/anaconda3/bin/conda")`
- Within virtualenvs and conda envs that carry the same name as the imported module. e.g. `~/anaconda/envs/nltk` for `import("nltk")`
- At the location of the Python binary discovered on the system PATH (i.e. `Sys.which("python")`)
- At customary locations for Python, e.g. `/usr/local/bin/python`, `/opt/local/bin/python...`

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at rstudio.github.io/reticulate/ • reticulate 1.20.0 • Updated: 2021-08

What is rmarkdown?



.Rmd files • Develop your code and ideas side-by-side in a single document. Run code as individual chunks or as an entire document.

Dynamic Documents • Knit together plots, tables, and results with narrative text. Render to a variety of formats like HTML, PDF, MS Word, or MS Powerpoint.

Reproducible Research • Upload, link to, or attach your report to share. Anyone can read or run your code to reproduce your work.

Workflow

- 1 Open a **new .Rmd file** in the RStudio IDE by going to **File > New File > R Markdown**.
- 2 **Embed code** in chunks. Run code by line, by chunk, or all at once.
- 3 **Write text** and add tables, figures, images, and citations. Format with Markdown syntax or the RStudio Visual Markdown Editor.
- 4 **Set output format(s) and options** in the YAML header. Customize themes or add parameters to execute or add interactivity with Shiny.
- 5 **Save and render** the whole document. Knit periodically to preview your work as you write.
- 6 **Share your work!**

Embed Code with knitr

CODE CHUNKS

Surround code chunks with `{{r}}` and `{{` or use the Insert Code Chunk button. Add a chunk label and/or chunk options inside the curly braces after **r**.

```
```{r chunk-label, include=FALSE}
summary(mtcars)
```
```

SET GLOBAL OPTIONS

Set options for the entire document in the first chunk.

```
```{r include=FALSE}
knitr::opts_chunk$message = FALSE
```
```

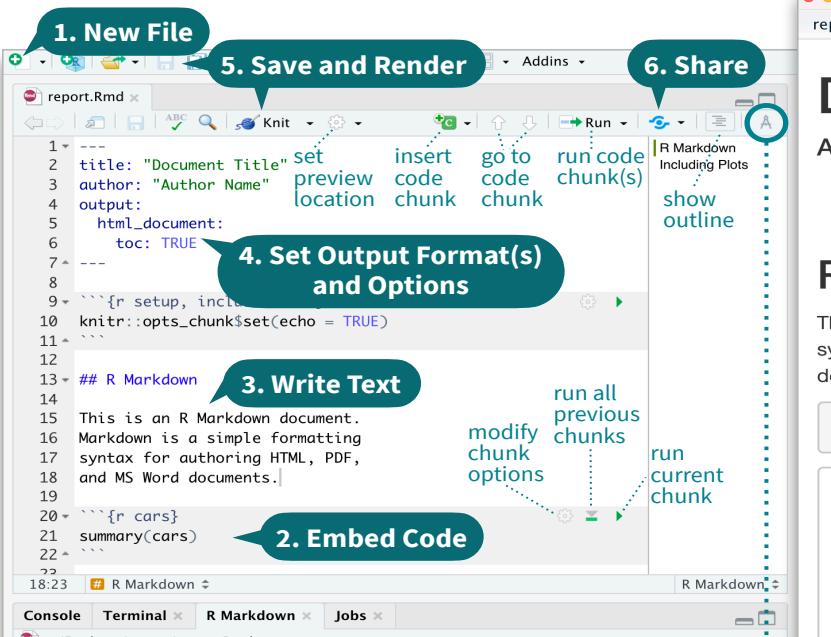
INLINE CODE

Insert `{{r <code>}}` into text sections. Code is evaluated at render and results appear as text.

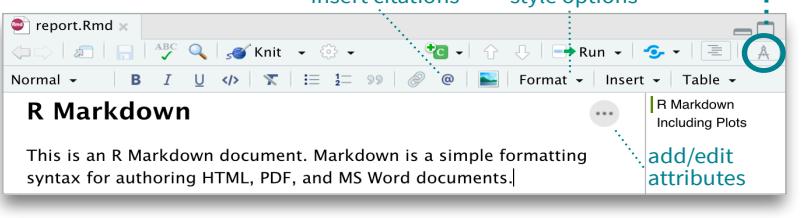
"Built with `{{r getRversion()}}`" --> "Built with 4.1.0"



SOURCE EDITOR



VISUAL EDITOR



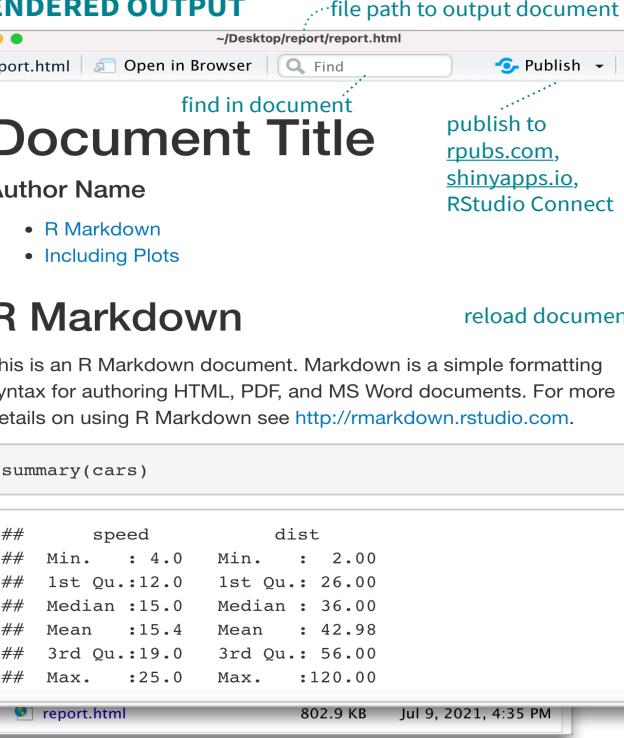
OPTION

DEFAULT EFFECTS

| | | |
|-----------------------------------|-------------|---|
| echo | TRUE | display code in output document |
| error | FALSE | TRUE (display error messages in doc)
FALSE (stop render when error occurs) |
| eval | TRUE | run code in chunk |
| include | TRUE | include chunk in doc after running |
| message | TRUE | display code messages in document |
| warning | TRUE | display code warnings in document |
| results | "markup" | "asis" (passthrough results)
"hide" (don't display results)
"hold" (put all results below all code) |
| fig.align | "default" | "left", "right", or "center" |
| fig.alt | NULL | alt text for a figure |
| fig.cap | NULL | figure caption as a character string |
| fig.path | "figure/{{" | prefix for generating figure file paths |
| fig.width & fig.height | 7 | plot dimensions in inches |
| out.width | | rescales output width, e.g. "75%", "300px" |
| collapse | FALSE | collapse all sources & output into a single block |
| comment | "##" | prefix for each line of results |
| child | NULL | files(s) to knit and then include |
| purl | TRUE | include or exclude a code chunk when extracting source code with knitr::purl() |

See more options and defaults by running `str(knitr::opts_chunk$get())`

RENDERED OUTPUT



Insert Citations

Create citations from a bibliography file, a Zotero library, or from DOI references.

BUILD YOUR BIBLIOGRAPHY

- Add BibTeX or CSL bibliographies to the YAML header.


```
---
title: "My Document"
bibliography: references.bib
link-citations: TRUE
---
```
- If Zotero is installed locally, your main library will automatically be available.
- Add citations by DOI by searching "from DOI" in the **Insert Citation** dialog.

INSERT CITATIONS

- Access the **Insert Citations** dialog in the Visual Editor by clicking the @ symbol in the toolbar or by clicking **Insert > Citation**.
- Add citations with markdown syntax by typing `[@cite]` or `@cite`.

Insert Tables

Output data frames as tables using `kable(data, caption)`.

```
```{r}
data <- faithful[1:4,]
knitr::kable(data,
 caption = "Table with kable")
```
```

Other table packages include `flextable`, `gt`, and `kableExtra`.

Write with Markdown

The syntax on the left renders as the output on the right.

Plain text.

End a line with two spaces to start a new paragraph.

Also end with a backslash\ to make a new line.

italics and **bold**

superscript²/subscript₂

~~strikethrough~~

escaped: *`_`\\`_`

endash: --, emdash: ---

Header 1

Header 2

...

Header 6

- unordered list

- item 2
- item 2a (indent 1 tab)
- item 2b

1. ordered list

2. item 2
- item 2a (indent 1 tab)
- item 2b

<link url>

[This is a link.](link url)

[This is another link][id].

At the end of the document:
[id]: link url

![Caption](image.png)
or![Caption][id2]

At the end of the document:
[id2]: image.png

`verbatim code`

````

multiple lines  
of verbatim code

> block quotes

equation: \$e^{i \pi} + 1 = 0\$

equation block:  
\$E = mc^2\$

horizontal rule:  
---

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1

### HTML Tables

# Results {.table}  
## Plots text

text  
## Tables  
more text

equation:  $e^{i\pi} + 1 = 0$

equation block:  
 $E = mc^2$

horizontal rule:  
---

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1

### Results

Plots  
Tables  
text

Use the document's YAML header to set an **output format** and customize it with **output options**.

```

```

```
title: "My Document"
author: "Author Name"
output:
 html_document:
 toc: TRUE
```

Indent format 2 characters,  
indent options 4 characters

OUTPUT FORMAT	CREATES
html_document	.html
pdf_document*	.pdf
word_document	Microsoft Word (.docx)
powerpoint_presentation	Microsoft Powerpoint (.pptx)
odt_document	OpenDocument Text
rtf_document	Rich Text Format
md_document	Markdown
github_document	Markdown for Github
ioslides_presentation	ioslides HTML slides
slidy_presentation	Slidy HTML slides
beamer_presentation*	Beamer slides

\* Requires LaTeX, use `tinytex::install_tinytex()`  
Also see `flexdashboard`, `bookdown`, `distill`, and `blogdown`.

## More Header Options

### PARAMETERS

Parameterize your documents to reuse with new inputs (e.g., data, values, etc.).

1. **Add parameters** in the header as sub-values of `params`.

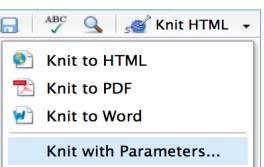
```

```

```
params:
 state: "hawaii"

```

```
```{r}
data <- df[, params$state]
summary(data)
```
```



### REUSABLE TEMPLATES

1. **Create a new package** with a `inst/rmarkdown/templates` directory.
  2. **Add a folder** containing `template.yaml` (below) and `skeleton.Rmd` (template contents).
- ```
---
```
- ```
name: "My Template"

```
3. **Install** the package to access template by going to **File > New R Markdown > From Template**.



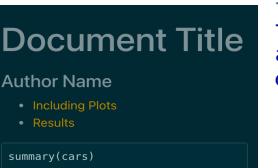
| IMPORTANT OPTIONS   | DESCRIPTION                                                                            | HTM     | PDF | MSW | MSP |
|---------------------|----------------------------------------------------------------------------------------|---------|-----|-----|-----|
| anchor_sections     | Show section anchors on mouse hover (TRUE or FALSE)                                    | X       |     |     |     |
| citation_package    | The LaTeX package to process citations ("default", "natbib", "biblatex")               | X       |     |     |     |
| code_download       | Give readers an option to download the .Rmd source code (TRUE or FALSE)                | X       |     |     |     |
| code_folding        | Let readers to toggle the display of R code ("none", "hide", or "show")                | X       |     |     |     |
| css                 | CSS or SCSS file to use to style document (e.g. "style.css")                           | X       |     |     |     |
| dev                 | Graphics device to use for figure output (e.g. "png", "pdf")                           | X X     |     |     |     |
| df_print            | Method for printing data frames ("default", "kable", "tibble", "paged")                | X X X X |     |     |     |
| fig_caption         | Should figures be rendered with captions (TRUE or FALSE)                               | X X X X |     |     |     |
| highlight           | Syntax highlighting ("tango", "pygments", "kate", "zenburn", "textmate")               | X X X   |     |     |     |
| includes            | File of content to place in doc ("in_header", "before_body", "after_body")             | X X     |     |     |     |
| keep_md             | Keep the Markdown .md file generated by knitting (TRUE or FALSE)                       | X X X X |     |     |     |
| keep_tex            | Keep the intermediate TEX file used to convert to PDF (TRUE or FALSE)                  | X       |     |     |     |
| latex_engine        | LaTeX engine for producing PDF output ("pdflatex", "xelatex", or "lualatex")           | X       |     |     |     |
| reference_docx/_doc | docx/pptx file containing styles to copy in the output (e.g. "file.docx", "file.pptx") | X X     |     |     |     |
| theme               | Theme options (see Bootswatch and Custom Themes below)                                 | X       |     |     |     |
| toc                 | Add a table of contents at start of document (TRUE or FALSE)                           | X X X X |     |     |     |
| toc_depth           | The lowest level of headings to add to table of contents (e.g. 2, 3)                   | X X X X |     |     |     |
| toc_float           | Float the table of contents to the left of the main document content (TRUE or FALSE)   | X       |     |     |     |

Use `?<output format>` to see all of a format's options, e.g. `?html_document`

### BOOTSWATCH THEMES

Customize HTML documents with Bootswatch themes from the `bslib` package using the theme output option.

Use `bslib::bootswatch_themes()` to list available themes.



```

```

```
title: "Document Title"
author: "Author Name"
output:
 html_document:
 theme:
 bootswatch: solar

```

### CUSTOM THEMES

Customize individual HTML elements using `bslib` variables. Use `?bs_theme` to see more variables.

```

```

```
output:
 html_document:
 theme:
 bg: "#121212"
 fg: "#E4E4E4"
 base_font:
 google: "Prompt"

```

More on `bslib` at [pkgs.rstudio.com/bslib/](https://pkgs.rstudio.com/bslib/).

### STYLING WITH CSS AND SCSS

Add CSS and SCSS to your document by adding a path to a file with the `css` option in the YAML header.

```

```

```
title: "My Document"
author: "Author Name"
output:
 html_document:
 css: "style.css"

```

Apply CSS styling by writing HTML tags directly or:

- Use markdown to apply style attributes inline.

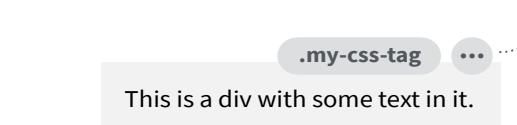
Bracketed Span  
A [green]{.my-color} word.

A green word.

Fenced Div  
:::{.my-color}  
All of these words  
are green.  
:::

All of these words  
are green.

- Use the Visual Editor. Go to **Format > Div/Span** and add CSS styling directly with Edit Attributes.



When you render a document, rmarkdown:

1. Runs the code and embeds results and text into an .md file with knitr.

2. Converts the .md file into the output format with Pandoc.



**Save**, then **Knit** to preview the document output. The resulting HTML/PDF/MS Word/etc. document will be created and saved in the same directory as the .Rmd file.

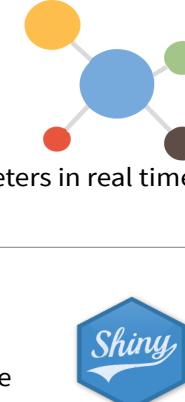
Use `rmarkdown::render()` to render/knit in the R console. See `?render` for available options.

## Share

### Publish on RStudio Connect

to share R Markdown documents securely, schedule automatic updates, and interact with parameters in real time.

[rstudio.com/products/connect/](https://rstudio.com/products/connect/)



### INTERACTIVITY

Turn your report into an interactive Shiny document in 4 steps:

1. Add `runtime: shiny` to the YAML header.
2. Call Shiny input functions to embed input objects.
3. Call Shiny render functions to embed reactive output.
4. Render with `rmarkdown::run()` or click **Run Document** in RStudio IDE.

```

```

```
output: html_document
runtime: shiny

```

```
```{r, echo = FALSE}
numericInput("n",
  "How many cars?", 5)

```

```
renderTable({
  head(cars, input$n)
})

```

How many cars?	
5	
speed	dist
1	4.00
2	4.00
3	7.00
4	7.00
5	8.00
	16.00

Also see Shiny Prerendered for better performance.

rstudio.com/rmarkdown/authoring_shiny_prerendered

Embed a complete app into your document with `shiny::shinyAppDir()`. More at bookdown.org/yihui/rmarkdown/shiny-embedded.html.



Install rphylopic

rphylopic allows you to add species' silhouettes from phylopic to ggplot2 or base plots:

[CRAN version](#)
install.packages("rphylopic")

[Development version](#)
install.packages("remotes")
remotes::install_github("sckott/rphylopic")
library('rphylopic')

uuid

Universally unique identifier (uuid) is a 128-bit number. It has 32 alphanumeric characters in the form of 8-4-4-4-12. Every silhouette has a uuid to uniquely identify it.

Find silhouettes

1. Work with names.

- **name_search**(text, options)[[1]]

Searches the uid code based on common name or taxonomy of a species. The options can be namebankID, type, names, root, uri.

- **name_get**(uid, options)

Get information on a name using the uid code. The options can be citationStart, html, namebankID, root.

- **name_images**(uid, options = 'credit')

Searches for different images for a taxonomic name.

- **name_taxonomy**(uid, options, as)

Returns taxonomic name based on uid code.

Options can be string, and as can be list, table, json.

- **name_taxonomy_many**(uid, options, as)

Returns taxonomic names for two or more concatenated (c()) uid codes.

- **name_taxonomy_sources**(uid)

Gives information on the sources for a name's taxonomy given a uid.

2. Work with uBio data

- **ubio_get**(namebankID)

Retrieve the uid code based on the namebankID number .

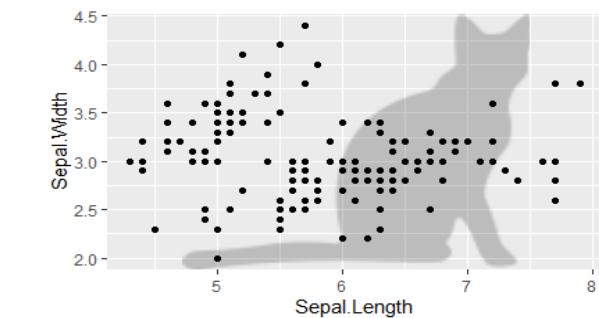
Plot silhouettes

1. Plot a silhouette behind a plot

- **ggplot**

```
library(ggplot2)
cat <- image_data("23cd6aa4-9587-4a2e-8e26-de42885004c9", size = 128)[[1]]
```

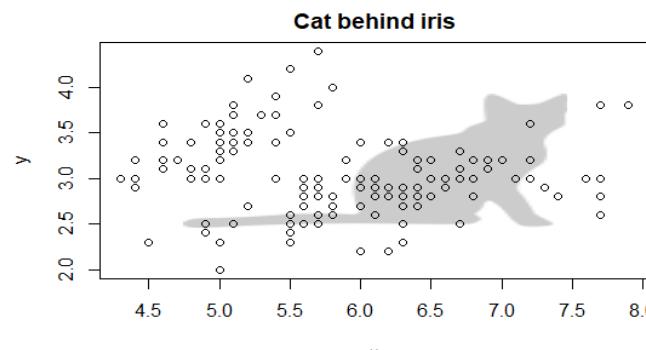
```
ggplot(data = iris,
       aes(x = Sepal.Length,
           y = Sepal.Width)) +
  geom_point() +
  add_phylopic(cat, alpha = 0.2)
```



- **Base plot**

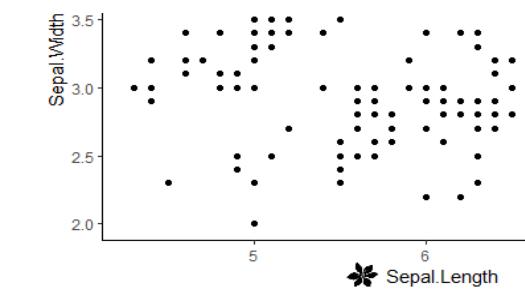
```
cat <- image_data("23cd6aa4-9587-4a2e-8e26-de42885004c9", size = 128)[[1]]
```

```
plot(1, 1,
     type = 'n',
     main = "Cat behind iris")
add_phylopic_base(cat,
                   x = 0.5,
                   y = 0.5,
                   ysize = 0.8,
                   alpha = 0.2)
```



2. Plot a silhouette anywhere in a plot

```
ggpubr::ggarrange(plot +
  add_phylopic(irisimg,
               alpha = 1,
               x = 0.43,
               y = 0.05,
               ysize = 0.06))
```

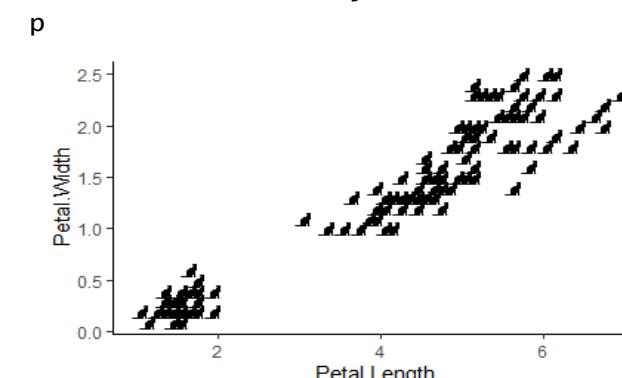


3. Plot silhouettes as points in a plot

- **ggplot2**

```
p <- ggplot(iris,
            aes(Petal.Length,
                Petal.Width)) +
  geom_blank() +
  theme_classic()

for (i in 1:nrow(iris)) {
  p <- p +
    add_phylopic(cat,
                 alpha = 1,
                 iris$Petal.Length[i],
                 iris$Petal.Width[i],
                 ysize = 0.2)
}
```



4. Save PNG file to disk

- Download a silhouette from <http://phylopic.org/> and save it in your working directory.
img <- png::readPNG("img.png")

5. Use silhouettes as icons in leaflet plots

```
library(leaflet)
data(quakes) ## this is a table
# get an image
cat <- image_data("23cd6aa4-9587-4a2e-8e26-de42885004c9", size = 128)[[1]]
# save to disk
catimg <- save_png(cat)
# make an icon. See ?makeIcon for more
# iconwidth is in pixels
cat_icon <- makeIcon(iconUrl = catimg,
                      iconwidth = 30)
# make the plot, just 7:10 rows
leaflet(data = quakes[7:10,]) %>%
  addTiles() %>%
  addMarkers(~long, ~lat,
             icon = cat_icon)
```

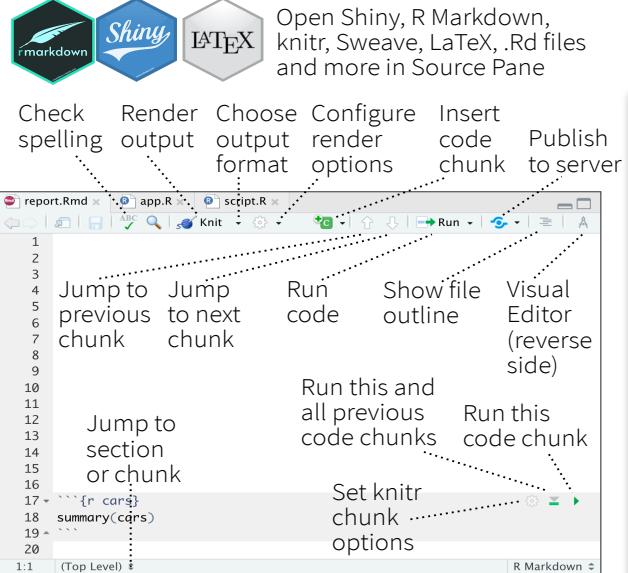


Citation

Don't forget to cite rphylopic. See how here:

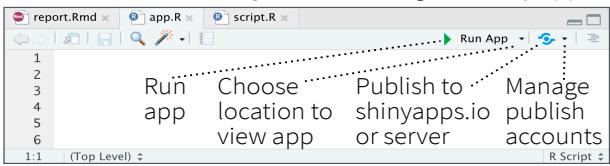
```
citation("rphylopic")
```

Documents and Apps



Access markdown guide at
Help > Markdown Quick Reference
See reverse side for more on **Visual Editor**

RStudio recognizes that files named **app.R**, **server.R**, **ui.R**, and **global.R** belong to a shiny app

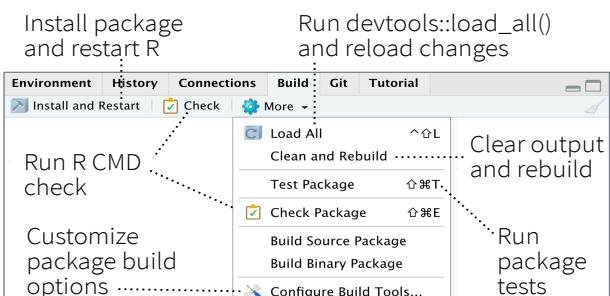


Package Development

Create a new package with
File > New Project > New Directory > R Package
Enable roxygen documentation with
Tools > Project Options > Build Tools

Roxygen guide at **Help > Roxygen Quick Reference**

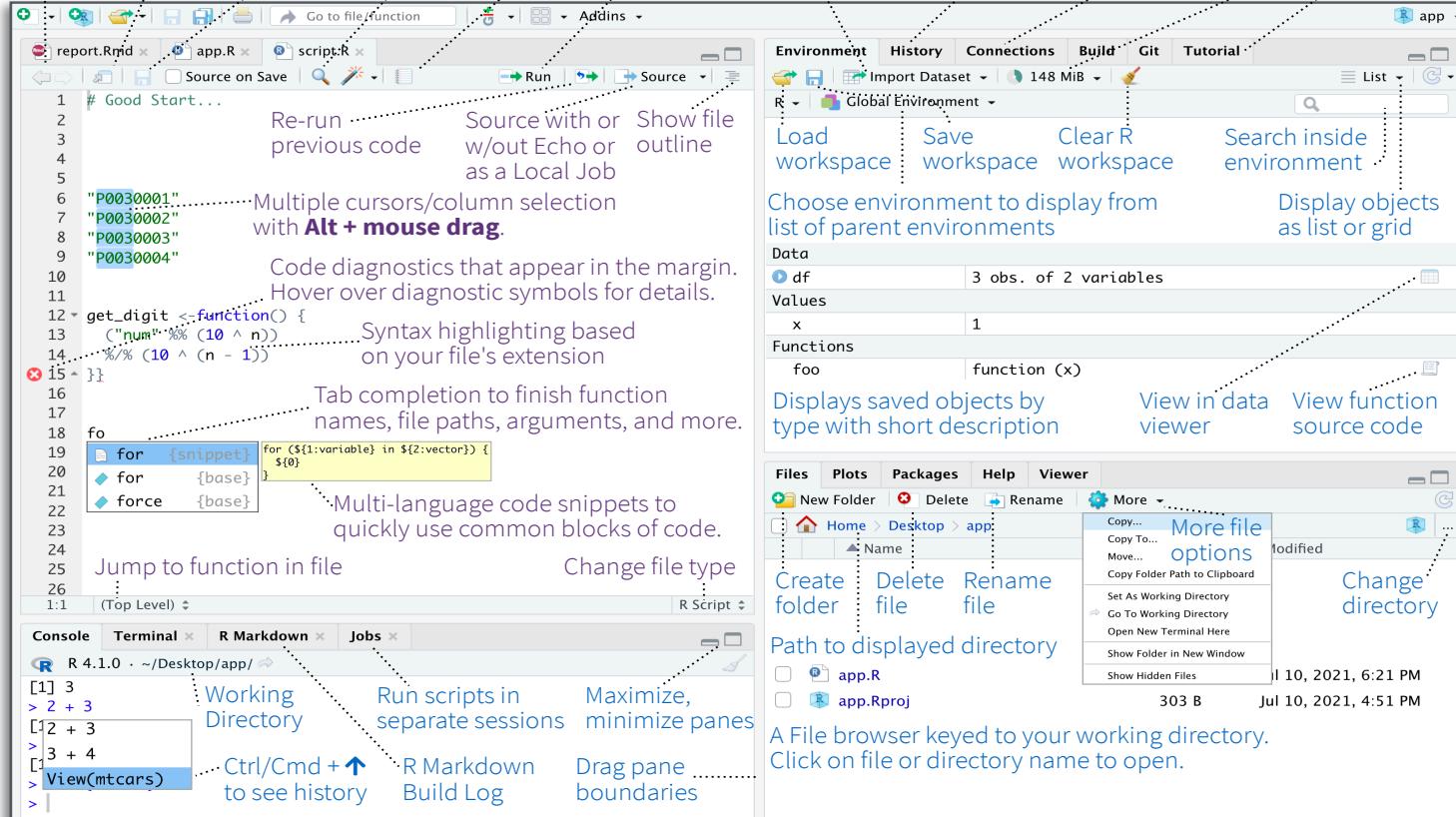
See package information in the **Build Tab**



R Studio

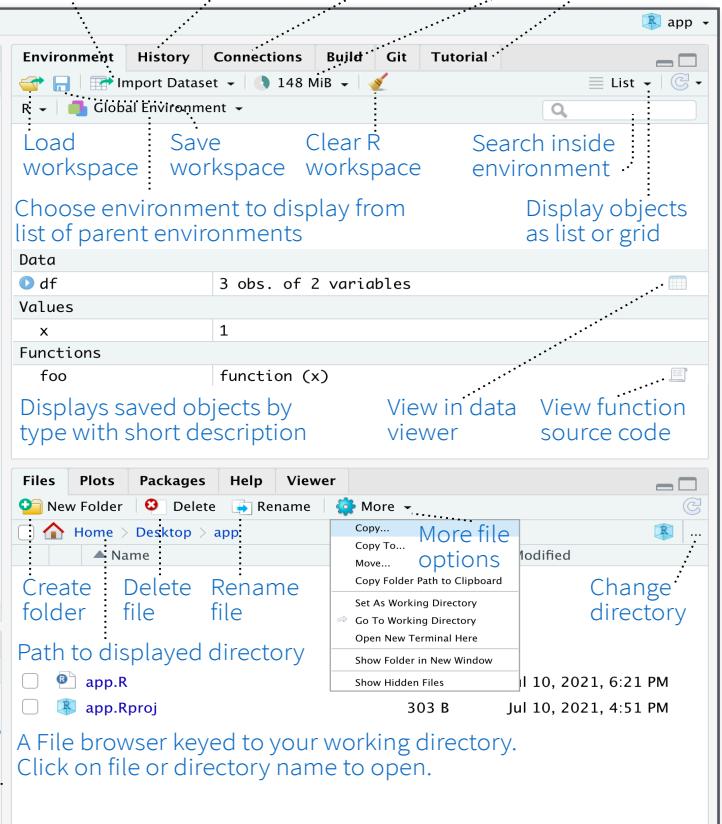
Source Editor

Navigate backwards/forwards, Open in new window, Save, Find and replace, Compile as notebook, Run selected code



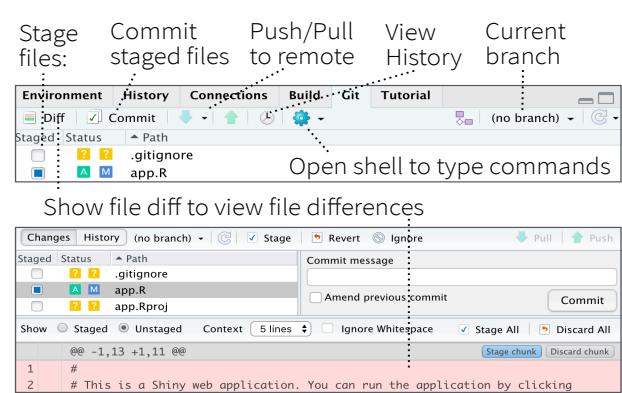
Tab Panes

Import data with wizard, History of past commands to run/copy, Manage external databases, View memory usage, R tutorials



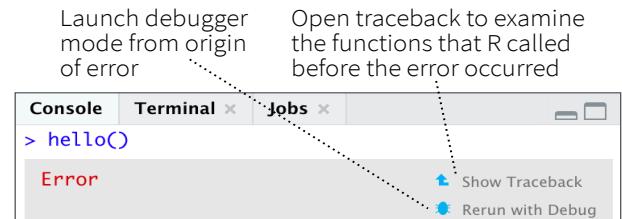
Version Control

Turn on at **Tools > Project Options > Git/SVN**
A Added, M Modified, D Deleted, R Renamed, U Untracked

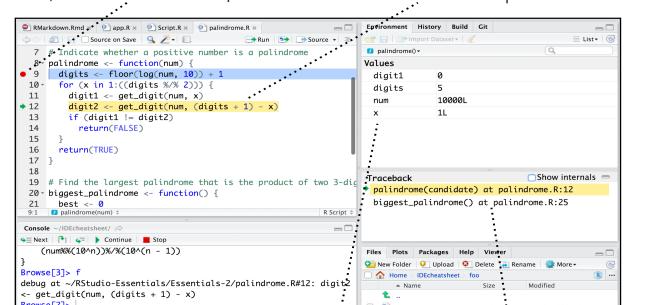


Debug Mode

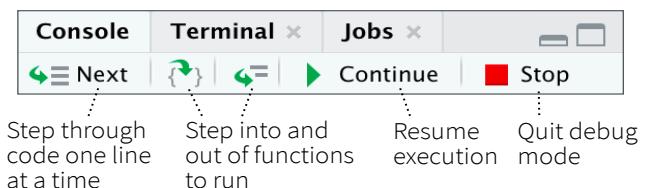
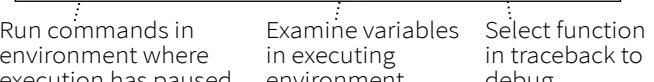
Use **debug()**, **browser()**, or a breakpoint and execute your code to open the debugger mode.



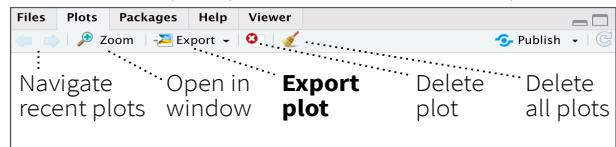
Click next to line number to add/remove a breakpoint.



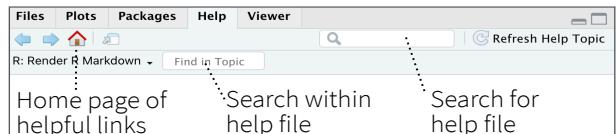
Run commands in environment where execution has paused



RStudio opens plots in a dedicated **Plots** pane



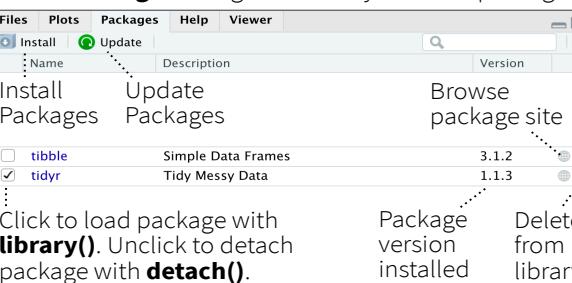
RStudio opens documentation in a dedicated **Help** pane



Viewer pane displays HTML content, such as Shiny apps, RMarkdown reports, and interactive visualizations



GUI Package manager lists every installed package



Filter rows by value or value range, Sort by values, Search for value



RUN CODE

Search command history
Interrupt current command
Clear console

Navigate Code

Go to File/Function

Write Code

Attempt completion

Insert <- (assignment operator)
Insert %>% (pipe operator)
(Un)Comment selection

MAKE PACKAGES

Load All (devtools)
Test Package (Desktop)
Document Package

Windows/Linux Mac

Ctrl+↑
Esc
Ctrl+L

Cmd+↑
Esc
Ctrl+L

DOCUMENTS AND APPS

Knit Document (knitr)
Insert chunk (Sweave & Knitr)
Run from start to current line

Ctrl+Shift+K
Ctrl+Alt+I
Ctrl+Alt+B

Cmd+Shift+K
Cmd+Option+I
Cmd+Option+B

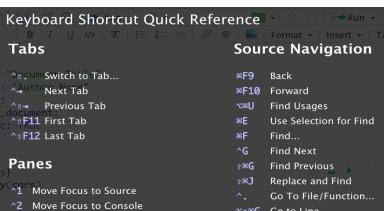
MORE KEYBOARD SHORTCUTS

Keyboard Shortcuts Help
Show Command Palette

Alt+Shift+K
Ctrl+Shift+P

Search for keyboard shortcuts with
Tools > Show Command Palette
or **Alt/Option + Shift + K**.

View the Keyboard Shortcut Quick
Reference with **Tools > Keyboard
Shortcuts** or **Alt/Option + Shift + K**.



Visual Editor

Block format

Check spelling Render output Choose output format Choose output location Insert code chunk Jump to previous chunk Jump to next chunk Run selected lines Publish to server Show file outline Back to Source Editor (front page)

Lists and block quotes Links Citations Images More formatting Insert blocks, citations, equations, and special characters Insert and edit tables R Markdown Including Plots File outline

Clear formatting Insert verbatim code Add/Edit attributes Set knitr chunk options Run this and all previous code chunks Run this code chunk

Jump to chunk or header # R Markdown

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents.

```
{r cars}
summary(cars)
```

Jump to chunk or header

Workbench

WHY RSTUDIO WORKBENCH?

Extend the open source server with a commercial license, support, and more:

- open and run multiple R sessions at once
- tune your resources to improve performance
- administrative tools for managing user sessions
- collaborate real-time with others in shared projects
- switch easily from one version of R to a different version
- integrate with your authentication, authorization, and audit practices
- work in the RStudio IDE, JupyterLab, Jupyter Notebooks, or VS Code

Download a free 45 day evaluation at

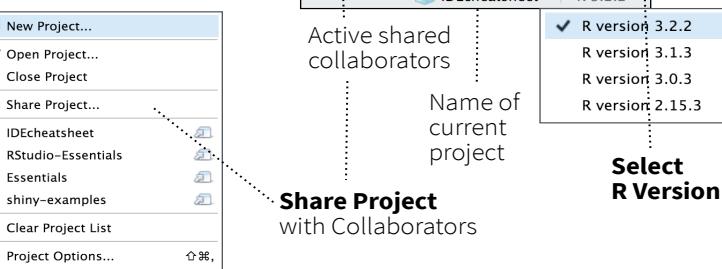
www.rstudio.com/products/workbench/evaluation/

Share Projects

File > New Project

RStudio saves the call history, workspace, and working directory associated with a project. It reloads each when you re-open a project.

Start **new R Session** in current project . . . Close R Session in project

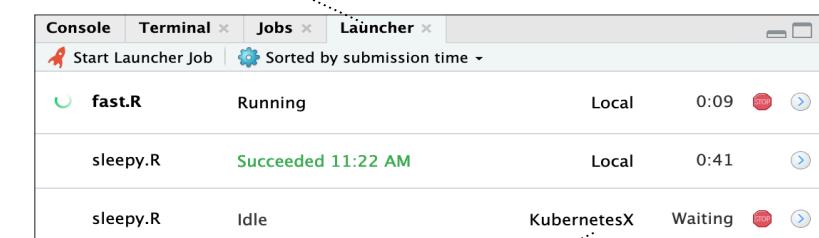


Active shared collaborators Name of current project Select R Version

Run Remote Jobs

Run R on remote clusters (Kubernetes/Slurm) via the Job Launcher

Monitor launcher jobs . . . Launch a job . . .



Run launcher jobs remotely

Optimal stratification

Given a sampling frame, SamplingStrata allows to optimize its stratification when designing a sampling survey, given precision constraints on target estimates.

Three different methods

The optimization can be run by indicating three different methods, on the basis of the following:

- A. if stratification variables are categorical (or reduced to) then the method is the "**atomic**";
- B. if stratification variables are continuous, then the method is the "**continuous**";
- C. if stratification variables are continuous, and there is spatial correlation among units in the sampling frame, then the required method is the "**spatial**".

A. Method "atomic"

Different steps:

1. define the sampling frame;
2. set precision constraints;
3. build atomic strata;
4. run optimization;
5. perform evaluation;
6. select the sample.

Sampling frame

```
library(SamplingStrata)
data("swissmunicipalities")
swissmunicipalities$id <- c(1:nrow(swissmunicipalities))
frame <- buildFrameDF(
  df = swissmunicipalities,
  id = "id",
  domainvalue = "REG",
  X = c("POPTOT", "HApoly"),
  Y = c("Surfacesbois", "Airind"))
```

Data on 2896 Swiss municipalities

Stratification variables

Target variables

Precision constraints

```
ndom <- length(unique(frame$domainvalue))
cv <- as.data.frame(list(
  DOM = rep("DOM1", ndom),
  CV1 = rep(0.10, ndom),
  CV2 = rep(0.10, ndom),
  domainvalue = c(1:ndom)))
```

10% of maximum expected CV

Atomic strata

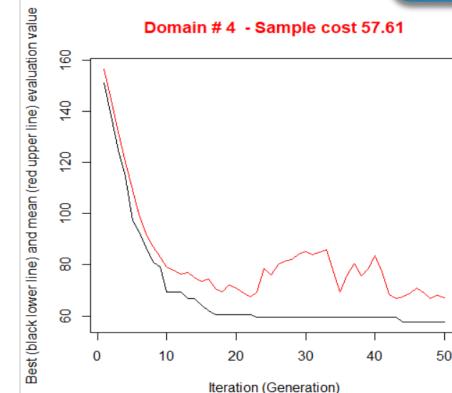
```
strata <- buildStrataDF(frame)
```

Optimization

```
solution <-
optimStrata(method="atomic",
            framesamp = frame,
            errors = cv,
            iter = 50,
            pops = 10)
```

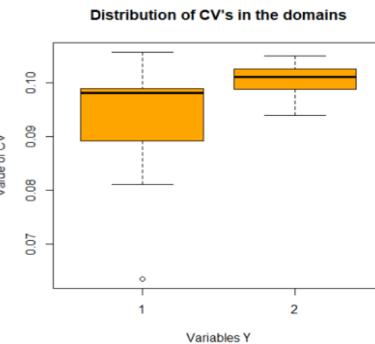
Number of iterations

Number of solutions per iteration



Evaluation

```
outstrata <- solution$aggr_strata
framenew <- solution$framenew
eval <- evalSolution(framenew, outstrata)
eval$coeff_var
```



Sample selection

```
s <- selectSample(framenew, outstrata)
head(s)
```

DOMAINVALUE	STRATO	ID	X1	X2	Y1	Y2	LABEL	WEIGHTS
1	1	2398	241	294	101	0	1	21.38462
2	1	2331	267	449	215	1	1	21.38462
3	1	2410	237	935	471	0	1	21.38462
4	1	2112	370	330	98	0	1	21.38462
5	1	2563	173	178	16	0	1	21.38462
6	1	2091	382	594	338	0	1	21.38462

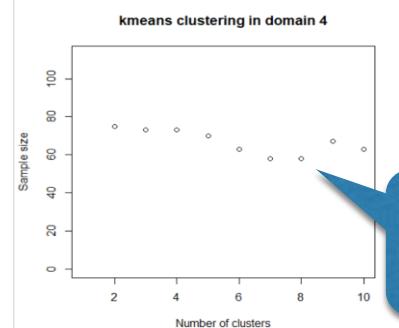
B. Method "continuous"

Same steps with the exception of strata building, not necessary.

Frame definition and precision constraints settings are done in the same way than in method "atomic". One more step is in determination of the most promising number of strata with kmeans clustering.

Kmeans clustering

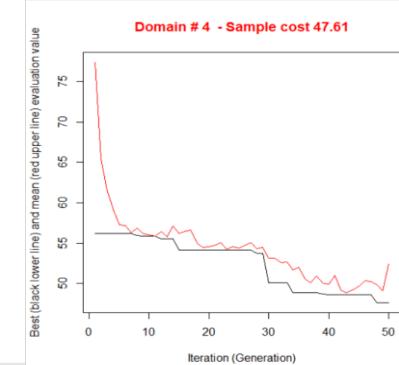
```
kmean <- KmeansSolution2(frame=frame,
                           errors=cv,
                           maxclusters = 10)
nstrat <- tapply(kmean$suggestions,
                  kmean$domainvalue,
                  FUN=function(x)
                    length(unique(x)))
sugg <- prepareSuggestion(
  kmean = kmean,
  frame = frame,
  nstrat = nstrat)
```



Optimization

```
solution <- optimStrata (
  method = "continuous",
  framesamp = frame,
  errors = cv,
  nStrata = nstrat,
  iter = 50,
  pops = 10,
  suggestions = sugg)
```

Suggestion prepared by kmeans clustering

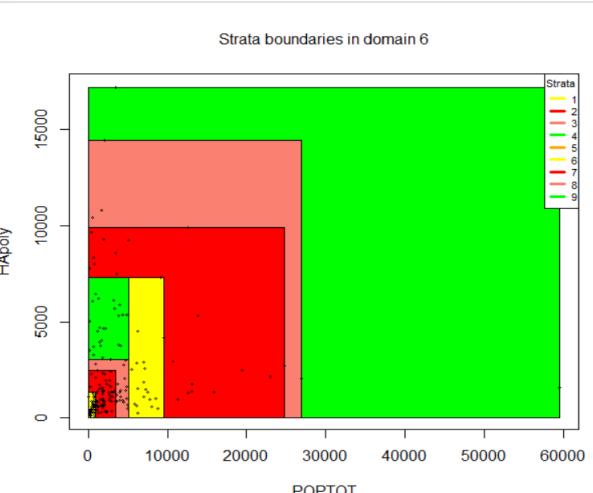


Evaluation

```
framenew <- solution$framenew
outstrata <- solution$aggr_strata
ss <- summaryStrata(framenew, outstrata)
head(ss)
```

Domain	Stratum	Population	Allocation	SamplingRate
1	1	278	13	0.047370
2	1	49	6	0.113769
3	1	70	5	0.070377
4	1	69	10	0.145075
5	1	33	6	0.173466
6	1	22	9	0.424133

```
plotStrata2d(framenew,
              outstrata,
              domain = 6,
              vars = c("X1", "X2"),
              labels = c("POPTOT", "HApoly"))
```



```
eval <-
evalSolution(framenew, outstrata)
eval$coeff_var
```

Sample selection

```
s <- selectSample(framenew, outstrata)
head(s)
```

DOMAINVALUE	STRATO	ID	X1	X2	Y1	Y2	LABEL	WEIGHTS
1	1	2398	241	294	101	0	1	21.38462
2	1	2331	267	449	215	1	1	21.38462
3	1	2410	237	935	471	0	1	21.38462
4	1	2112	370	330	98	0	1	21.38462
5	1	2563	173	178	16	0	1	21.38462
6	1	2091	382	594	338	0	1	21.38462

geo-referenced and there is spatial correlation among them, it is possible to apply the "spatial" method in the optimization of the frame stratification.

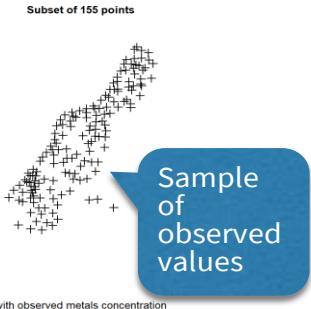
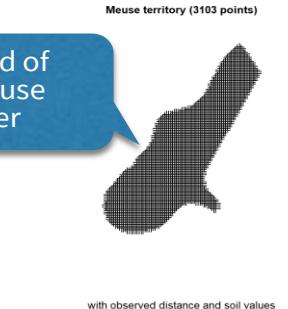
Different steps:

1. perform a preliminary spatial analysis and fit spatial models on target variables
2. define the sampling frame and add predicted values, prediction errors and coordinates;
3. set precision constraints;
4. run optimization;
5. select the sample.

Spatial analysis

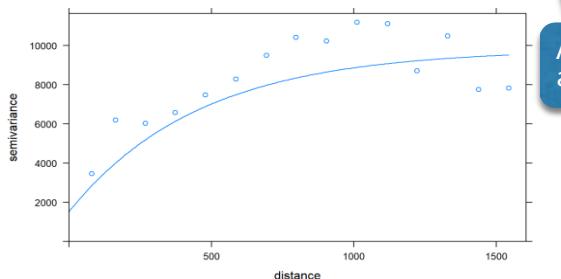
We make use of the «Meuse river» datasets, reporting measures of 4 metals concentration.

```
library(sp)
# locations (155 observed points)
data("meuse")
# grid of points (3103)
data("meuse.grid")
meuse.grid$id <- c(1:nrow(meuse.grid))
coordinates(meuse)<-c('x','y')
coordinates(meuse.grid)<-c('x','y')
```



Sample of observed values

```
library(gstat)
library(automap)
v <- variogram(lead~dist+soil,data=meuse)
fit.vgm.lead <- autofitVariogram(
    lead ~dist+soil,meuse,model="Exp")
plot(v, fit.vgm.lead$var_model)
```



```
0,lead.kr[1]$vari.pred)
lead.var <- ifelse(lead.kr[2]$var1.var < 0,
0,lead.kr[2]$var1.var)
```

Sampling frame

```
df <- as.data.frame(list(
  dom=rep(1,nrow(meuse.grid)),
  lead.pred=lead.pred,
  lead.var=lead.var,
  lon=meuse.grid$x,
  lat=meuse.grid$y,
  id=c(1:nrow(meuse.grid))))
frame <- buildFrameSpatial(df=df,
  id="id",
  X=c("lead.pred"),
  Y=c("lead.pred"),
  variance=c ("lead.var"),
  lon="lon",
  lat="lat",
  domainvalue = "dom")
```

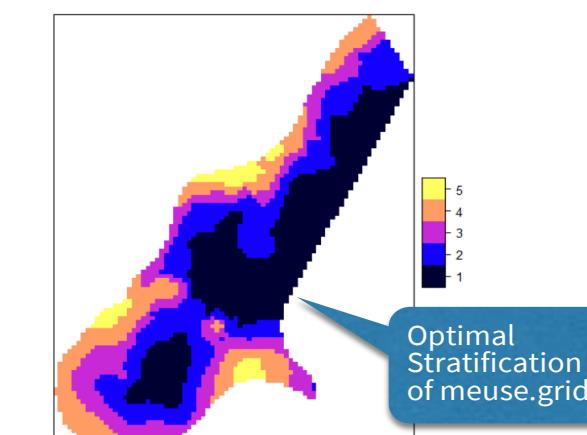
Precision constraints

```
cv2 <- as.data.frame(list(
  DOM=rep("DOM1",1),
  CV1=rep(0.05,1),
  domainvalue=c(1:1) ))
```

Optimization

```
solution <- optimStrata(method="spatial",
  errors=cv2, framesamp=frame, iter=25,
  nStrata=5, fitting=1, kappa=1,
  range=fit.vgm.lead$var_model$range[2])
```

```
framenew <- solution$framenew
outstrata <- solution$aggr_strata
frameres <- SpatialPixelsDataFrame(
  points=framenew[c("LON", "LAT")],
  data=framenew)
frameres$LABEL <-
  as.factor(frameres$LABEL)
spplot(frameres,c("LABEL"),
  col.regions=bpy.colors(5))
```



to calculate correctly the variance of target variables in strata, we can make use of models. When applying methods 'atomic' and 'continuous', it is possible to declare linear or log-linear models linking each target variable to one co-variate available in the sampling frame.

Consider the case with 'swissmunicipalities' dataset. Suppose that for all units we only have values for POPTOT and HApolY, while only on a subset (500) of it the values for Surfacesbois and Airbat are also available. We fit the following models:

```
k <- sample(c(1:2896),500)
s <- swissmunicipalities[k,]
Airind_POPTOT <-
  lm(Airind~POPTOT, data=s)
Bois_HApoly <-
  lm(Surfacesbois~HApoly, data=s)
```

For both models we calculate heteroscedasticity indexes and variance:

```
airind <-
  computeGamma(Airind_POPTOT$residuals,
  s$POPTOT,nbins = 14)
airind
# gamma sigma r.square
# 0.59235109 0.06794055 0.87070106
bois <-
  computeGamma(Bois_HApoly$residuals,
  s$HApoly,nbins = 14)
bois
# gamma sigma r.square
# 0.8547931 0.4483606 0.9732122 )
```

We can now instantiate the values in the 'model' dataframe:

```
model <- NULL
model$beta[1] <-
  Airind_POPTOT$coefficients[2]
model$sig2[1] <- airind[2]^2
model$type[1] <- "linear"
model$gamma[1] <- airind[1]
model$beta[2] <-
  Bois_HApoly$coefficients[2]
model$sig2[2] <- bois[2]^2
model$type[2] <- "linear"
model$gamma[2] <- bois[1]
model <- as.data.frame(model)
model
# beta      sig2      type      gamma
# 0.01109583 0.1708807 linear 0.4703953
# 0.26068155 0.2010272 linear 0.8547931
```

Co-variates as both X's and Y's

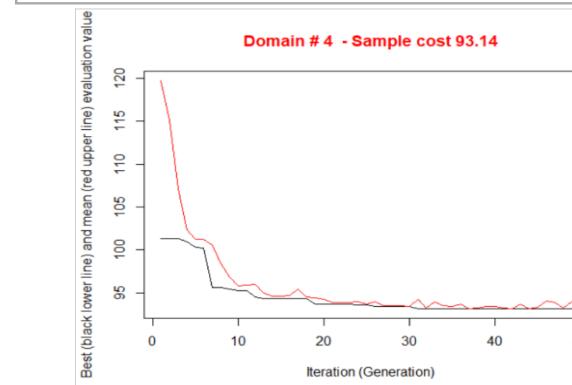
```
frame$airind <-
  swissmunicipalities$Airind
frame$surfacesbois <-
  swissmunicipalities$Surfacesbois
```

Optimization

With the same precision constraints of 10% for both target variables we run the optimization step:

```
solution <-
  optimStrata(
    method = "continuous",
    errors = cv,
    framesamp = frame,
    model = model,
    nStrata = rep(5,7),
    iter = 50,
    pops = 10)
```

'model' dataframe previously defined



Evaluation

```
framenew <- solution$framenew
outstrata <- solution$aggr_strata
framenew$Y3 <- framenew$AIRIND
framenew$Y4 <- framenew$SURFACESBOIS
val <- evalSolution(framenew,outstrata)
val$coeff_var
# CV1   CV2   CV3   CV4   dom
# 0.0107 0.0706 0.0316 0.0603 DOM1
# 0.0073 0.0364 0.0220 0.0426 DOM2
# 0.0062 0.0252 0.0253 0.0332 DOM3
# 0.0071 0.0328 0.0303 0.0572 DOM4
# 0.0055 0.0646 0.0171 0.0541 DOM5
# 0.0037 0.0745 0.0173 0.0606 DOM6
# 0.0036 0.0753 0.0145 0.0541 DOM7
```

Notice that both the CV's of the co-variates (CV1 and CV2) and the CV's of the real target variables (CV3 and CV4) are compliant to the 10% precision constraints.

Introduction

This guide aims to familiarise SAS users with R.
R examples make use of tidyverse collection of packages.

Install tidyverse:
library(tidyverse)

Attach tidyverse packages for use:

R data here in 'data frames', and occasionally vectors (via `c()`)

Other R structures (lists, matrices...) are not explored here.

Keyboard shortcuts: `<-` `Alt + -` `%>%` `Ctrl + Shift + m`

Datasets; drop, keep & rename variables

```
data new_data;
set old_data;
run;
```

```
new_data <- old_data
```

```
data new_data (keep=id);
set old_data (drop=job_title) ;
run;
```

```
new_data <- old_data %>%
select(-job_title) %>%
select(id)
```

```
data new_data (drop= temp: );
set old_data;
run;
```

```
new_data <- old_data %>%
select(-starts_with("temp"))
C.f. contains( ), ends_with()
```

```
data new_data;
set old_data;
rename old_name = new_name;
run;
```

```
new_data <- old_data %>%
rename(new_name = old_name)
```

Note order differs

Conditional filtering

```
data new_data;
set old_data;
if Sex = "M";
run;
```

```
new_data <- old_data %>%
filter(Sex == "M")
```

```
data new_data;
set old_data;
if year in (2010,2011,2012);
run;
```

```
new_data <- old_data %>%
filter(year %in% c(2010,2011,2012))
```

```
data new_data;
set old_data;
by id ;
if first.id ;
run;
```

```
new_data <- old_data %>%
group_by( id ) %>%
slice(1)
```

Could use slice(n()) for last

```
data new_data;
set old_data;
if dob > "25APR1990" d;
run;
```

```
new_data <- old_data %>%
filter(dob > as.Date("1990-04-25"))
```

New variables, conditional editing

```
data new_data;
set old_data;
total_income = wages + benefits ;
run;
```

```
new_data <- old_data %>%
mutate(total_income = wages + benefits)
```

```
data new_data;
set old_data;
if hours > 30 then full_time = "Y";
else full_time = "N";
run;
```

```
new_data <- old_data %>%
mutate(full_time = if_else(hours > 30 , "Y" , "N"))
```

```
data new_data;
set old_data;
if temp > 20 then weather = "Warm";
else if temp > 10 then weather = "Mild";
else weather = "Cold";
run;
```

```
new_data <- old_data %>%
mutate(weather = case_when(
  temp > 20 ~ "Warm",
  temp > 10 ~ "Mild",
  TRUE ~ "Cold" ))
```

Counting and Summarising

```
proc freq data = old_data ;
table job_type ;
run;
```

```
old_data %>%
count(job_type)
```

For percent, add:
%>% mutate(percent = n*100/sum(n))

```
proc freq data = old_data ;
table job_type*region ;
run;
```

```
old_data %>%
count(job_type , region )
```

```
proc summary data = old_data nway ;
class job_type region ;
output out = new_data ;
run;
```

```
new_data <- old_data %>%
group_by( job_type , region ) %>%
summarise( Count = n() )
```

Equivalent without nway not trivially produced

```
proc summary data = old_data nway ;
class job_type region ;
var salary ;
output out = new_data
sum( salary ) = total_salaries ;
run;
```

```
new_data <- old_data %>%
group_by(job_type , region ) %>%
summarise( total_salaries = sum( salary ) ,
Count = n() )
```

Lots of summary functions in both languages

Swap summarise() for mutate() to add summary data to original data

Combining datasets

```
data new_data ;
set data_1 data_2 ;
run;
```

```
new_data <- bind_rows( data_1 , data_2 )
```

C.f. rbind() which produces error if columns are not identical

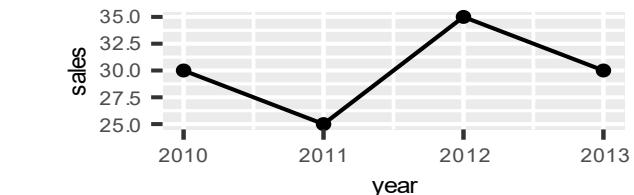
```
data new_data ;
merge data_1 (in= in_1) data_2 ;
by id ;
if in_1 ;
run;
```

```
new_data <- left_join( data_1 , data_2 , by = "id" )
```

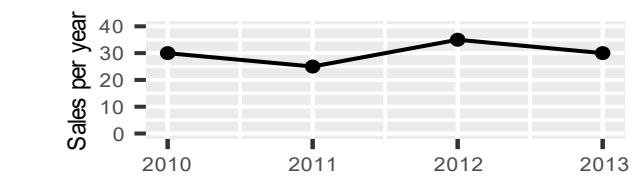
C.f. full_join() , right_join() , inner_join()

Some plotting in R

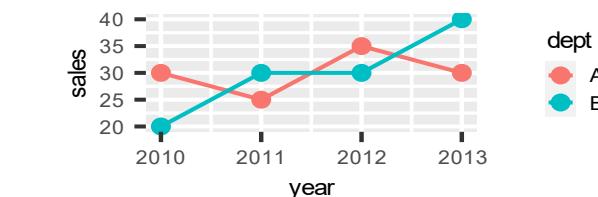
```
ggplot( my_data , aes( year , sales ) ) +
geom_point( ) + geom_line()
```



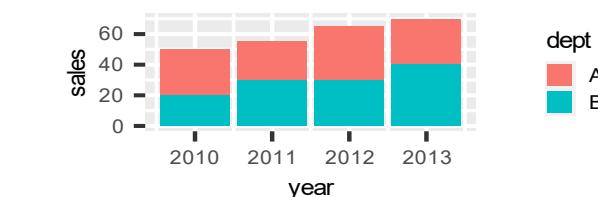
```
ggplot( my_data , aes( year , sales ) ) +
geom_point( ) + geom_line( ) + ylim(0, 40) +
labs(x = "" , y = "Sales per year")
```



```
ggplot(my_data, aes( year, sales, colour = dept ) ) +
geom_point( ) + geom_line()
```

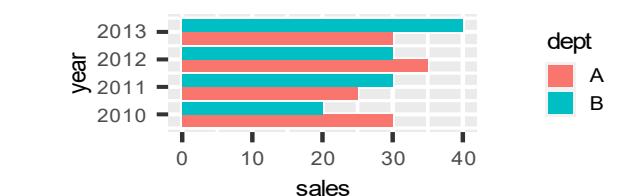


```
ggplot( my_data , aes( year, sales, fill = dept ) ) +
geom_col()
```



Note 'colour' for lines & points, 'fill' for shapes

```
ggplot( my_data , aes( year, sales, fill = dept ) ) +
geom_col( position = "dodge" ) + coord_flip()
```



C.f. position = "fill" for 100% stacked bars/cols

```

proc sort data=old_data out=new_data;
  by id descending income ;
run;

proc sort data=old_data nodup;
  by id job_type;
run;


Note nodup relies on adjacency of duplicate rows, distinct( ) does not



proc sort data=old_data nodupkey;
  by id ;
run;

data new_data;
  set old_data;
  by id descending income ;
  if first.id ;
run;

data new_data;
  set old_data;
  prev_id=lag(id);
run;


C.f. lead( ) for subsequent rows



data new_data;
  set old_data;
  by id;
  counter+1;
  if first.id then counter = 1;
run;

```

Converting and Rounding

```

data new_data;
  set old_data ;
  num_var = input("5" , 8. );
  text_var = put( 5 , 8. );
run;

data new_data ;
  set old_data;
  nearest_5 = round( x , 5 )
  two_decimals = round( x , 0.01 )
run;

```

Creating functions to modify datasets

```

%macro add_variable(dataset_name);
data &dataset_name;
  set &dataset_name;
  new_variable = 1;
run;
%mend;
%add_variable( my_data );

add_variable <- function( dataset_name ){
  dataset_name <- dataset_name %>%
    mutate(new_variable = 1)
  return( dataset_name )
}
my_data <- add_variable( my_data )



Note SAS can modify within the macro,  
whereas R creates a copy within the function


```

```

data new_data;
  set old_data;
  if find(job_title , "Health" );
run;

data new_data;
  set old_data;
  if job_title ==: "Health" ;
run;

data new_data;
  set old_data;
  substring = substr( big_string , 3 , 4 );
run;

data new_data;
  set old_data;
  address = tranwrd( address , "Street" , "St" );
run;

data new_data;
  set old_data;
  full_name = catx(" " , first_name , surname );
run;

data new_data;
  set old_data;
  first_word = scan( sentence , 1 );
run;

data new_data;
  set old_data;
  house_number = compress( address , , "dk" );
run;


R example preserves punctuation at the end of words, SAS doesn't



Wide range of regexps in both languages, this example extracts digits only


```

File operations

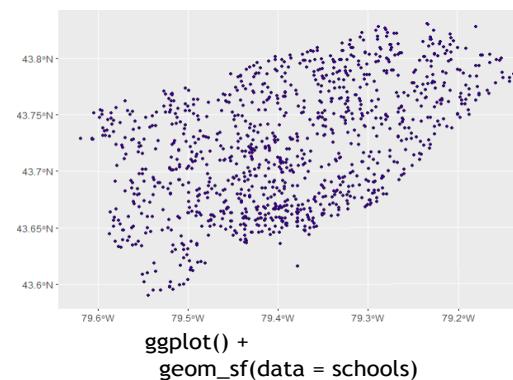
<p>Operate in 'Work' library. Use libname to define file locations</p> <pre> libname library_name "file_location"; data library_name.saved_data; set data_in_use; run; </pre> <pre> libname library_name "file_location"; data data_in_use ; set library_name.saved_data ; run; </pre> <pre> proc export data = my_data outfile = "my_file.csv" dbms = csv replace; run; </pre> <pre> proc import datafile = "my_file.csv" out = my_data dbms = csv; run; </pre>	<p>Operate in a particular 'working directory' (identify using getwd()) Move to other locations using setwd()</p> <pre> saveRDS(data_in_use , file="file_location/saved_data.rds") or setwd("file_location") saveRDS(data_in_use , file = "saved_data.rds") </pre> <pre> data_in_use <- readRDS("file_location/saved_data.rds") or setwd("file_location") data_in_use <- readRDS("saved_data.rds") </pre> <pre> write_csv(my_data , "my_file.csv") </pre> <pre> my_data <- read_csv("my_file.csv") </pre>
---	--

Both examples assume column headers in csv file



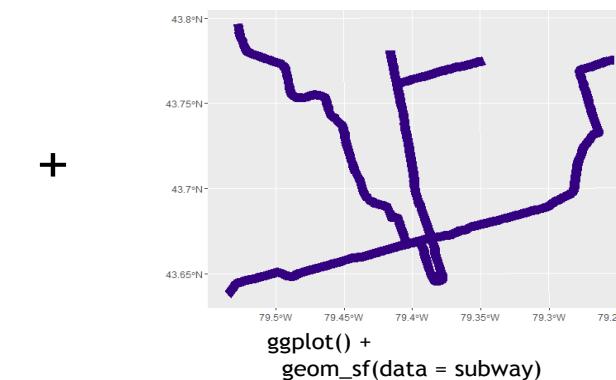
Geometric confirmation

- st_contains(x, y, ...) Identifies if y is within x (i.e. point within polygon)
- st_covered_by(x, y, ...) Identifies if x is completely within y (i.e. polygon completely within polygon)
- st_covers(x, y, ...) Identifies if any point from x is outside of y (i.e. polygon outside polygon)
- st_crosses(x, y, ...) Identifies if any geometry of x have commonalities with y
- st_disjoint(x, y, ...) Identifies when geometries from x do not share space with y
- st_equals(x, y, ...) Identifies if x and y share the same geometry
- st_intersects(x, y, ...) Identifies if x and y geometry share any space
- st_overlaps(x, y, ...) Identifies if geometries of x and y share space, are of the same dimension, but are not completely contained by each other
- st_touches(x, y, ...) Identifies if geometries of x and y share a common point but their interiors do not intersect
- st_within(x, y, ...) Identifies if x is in a specified distance to y



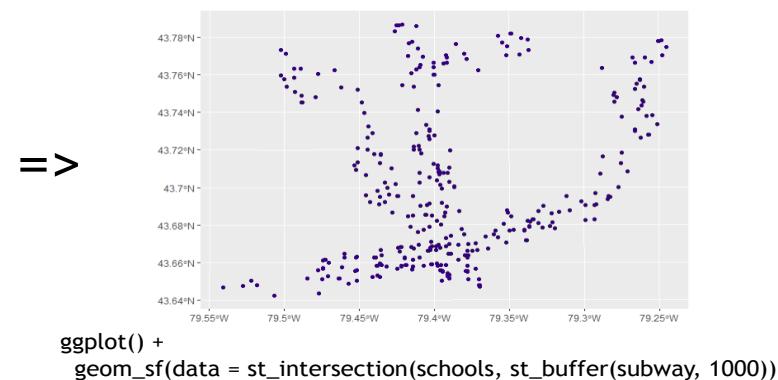
Geometric operations

- st_boundary(x) Creates a polygon that encompasses the full extent of the geometry
- st_buffer(x, dist, nQuadSegs) Creates a polygon covering all points of the geometry within a given distance
- st_centroid(x, ..., of_largest_polygon) Creates a point at the geometric centre of the geometry
- st_convex_hull(x) Creates geometry that represents the minimum convex geometry of x
- st_line_merge(x) Creates linestring geometry from sewing multi linestring geometry together
- st_node(x) Creates nodes on overlapping geometry where nodes do not exist
- st_point_on_surface(x) Creates a point that is guaranteed to fall on the surface of the geometry
- st_polygonize(x) Creates polygon geometry from linestring geometry
- st_segmentize(x, dfMaxLength, ...) Creates linestring geometry from x based on a specified length
- st_simplify(x, preserveTopology, dTolerance) Creates a simplified version of the geometry based on a specified tolerance



Geometry creation

- st_triangulate(x, dTolerance, bOnlyEdges) Creates polygon geometry as triangles from point geometry
- st_voronoi(x, envelope, dTolerance, bOnlyEdges) Creates polygon geometry covering the envelope of x, with x at the centre of the geometry
- st_point(x, c(numeric vector), dim = "XYZ") Creating point geometry from numeric values
- st_multipoint(x = matrix(numeric values in rows), dim = "XYZ") Creating multi point geometry from numeric values
- st_linestring(x = matrix(numeric values in rows), dim = "XYZ") Creating linestring geometry from numeric values
- st_multilinestring(x = list(numeric matrices in rows), dim = "XYZ") Creating multi linestring geometry from numeric values
- st_polygon(x = list(numeric matrices in rows), dim = "XYZ") Creating polygon geometry from numeric values
- st_multipolygon(x = list(numeric matrices in rows), dim = "XYZ") Creating multi polygon geometry from numeric values





Geometry operations

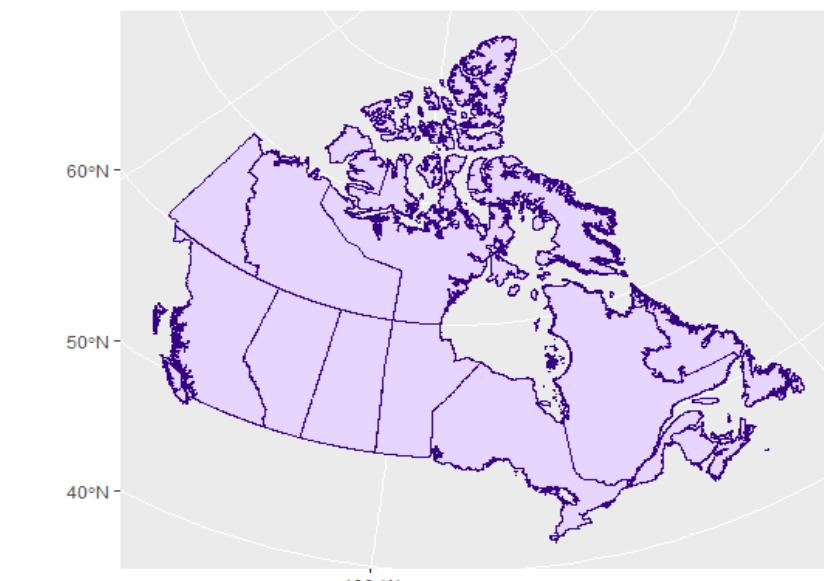
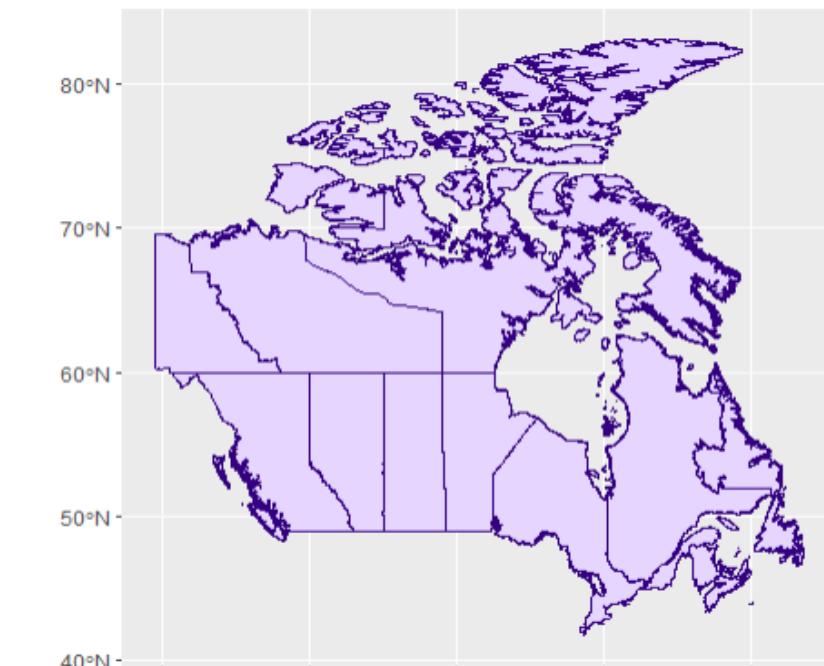
- st_contains(x, y, ...)** Identifies if y is within x (i.e. point within polygon)
- st_crop(x, y, ..., xmin, ymin, xmax, ymax)** Creates geometry of x that intersects a specified rectangle
- st_difference(x, y)** Creates geometry from x that does not intersect with y
- st_intersection(x, y)** Creates geometry of the shared portion of x and y
- st_sym_difference(x, y)** Creates geometry representing portions of x and y that do not intersect
- st_snap(x, y, tolerance)** Snap nodes from geometry x to geometry y
- st_union(x, y, ..., by_feature)** Creates multiple geometries into a single geometry, consisting of all geometry elements

Misc operations

- st_as_sf(x, ...)** Create a sf object from a non-geospatial tabular data frame
- st_cast(x, to, ...)** Change x geometry to a different geometry type
- st_coordinates(x, ...)** Creates a matrix of coordinate values from x
- st_crs(x, ...)** Identifies the coordinate reference system of x
- st_join(x, y, join, FUN, suffix, ...)** Performs a spatial left or inner join between x and y
- st_make_grid(x, cellsize, offset, n, crs, what)** Creates rectangular grid geometry over the bounding box of x
- st_nearest_feature(x, y)** Creates an index of the closest feature between x and y
- st_nearest_points(x, y, ...)** Returns the closest point between x and y
- st_read(dsn, layer, ...)** Read file or database vector dataset as a sf object
- st_transform(x, crs, ...)** Convert coordinates of x to a different coordinate reference system

Geometric measurement

- st_area(x)** Calculate the surface area of a polygon geometry based on the current coordinate reference system
- st_distance(x, y, ..., dist_fun, by_element, which)** Calculates the 2D distance between x and y based on the current coordinate system
- st_length(x)** Calculates the 2D length of a geometry based on the current coordinate system



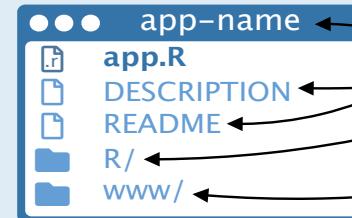
Building an App

A **Shiny** app is a web page (**ui**) connected to a computer running a live R session (**server**).



Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

Save your template as **app.R**. Keep your app in a directory along with optional extra files.



Launch apps stored in a directory with **runApp(<path to directory>)**.

To generate the template, type **shinyapp** and press **Tab** in the RStudio IDE or go to **File > New Project > New Directory > Shiny Web Application**

```
# app.R
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

server <- function(input, output, session) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

shinyApp(ui = ui, server = server)
```

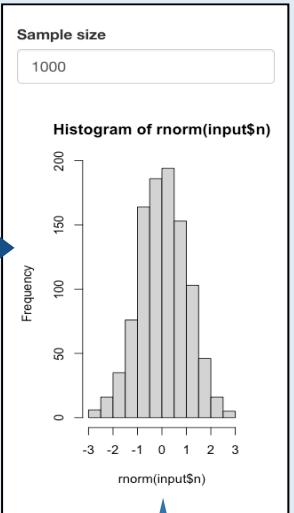
Customize the UI with Layout Functions

Add Inputs with *Input() functions

Add Outputs with *Output() functions

Wrap code in render*() functions before saving to output

Refer to UI inputs with input\$<id> and outputs with output\$<id>



Call shinyApp() to combine ui and server into an interactive app!

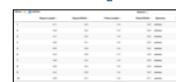
See annotated examples of Shiny apps by running **runExample(<example name>)**. Run **runExample()** with no arguments for a list of example names.

Share

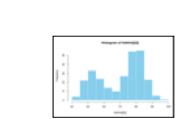
Share your app in three ways:

1. **Host it on shinyapps.io**, a cloud based service from RStudio. To deploy Shiny apps:
 - Create a free or professional account at shinyapps.io
 - Click the Publish icon in RStudio IDE, or run: `rsconnect::deployApp("<path to directory>")`
2. **Purchase RStudio Connect**, a publishing platform for R and Python. rstudio.com/products/connect/
3. **Build your own Shiny Server** rstudio.com/products/shiny/shiny-server/

Outputs



DT::renderDataTable(expr, options, searchDelay, callback, escape, env, quoted, outputArgs)

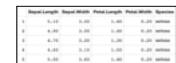


renderImage(expr, env, quoted, deleteFile, outputArgs)

renderPlot(expr, width, height, res, ..., alt, env, quoted, execOnResize, outputArgs)



renderPrint(expr, env, quoted, width, outputArgs)



renderTable(expr, striped, hover, bordered, spacing, width, align, rownames, colnames, digits, na, ..., env, quoted, outputArgs)



renderText(expr, env, quoted, outputArgs, sep)

renderUI(expr, env, quoted, outputArgs)

dataTableOutput(outputId)

imageOutput(outputId, width, height, click, dblclick, hover, brush, inline)

plotOutput(outputId, width, height, click, dblclick, hover, brush, inline)

verbatimTextOutput(outputId, placeholder)

tableOutput(outputId)

textOutput(outputId, container, inline)

uiOutput(outputId, inline, container, ...)

htmlOutput(outputId, inline, container, ...)

Inputs

Collect values from the user.

Access the current value of an input object with **input\$<inputId>**. Input values are **reactive**.

Action

actionButton(inputId, label, icon, width, ...)

Link

- Choice 1
- Choice 2
- Choice 3

checkboxInput(inputId, label, value, width)

dateInput(inputId, label, value, min, max, format, startview, weekstart, language, width, autoclose, datesdisabled, daysofweekdisabled)

dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator, width, autoclose)

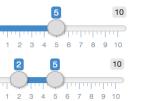
fileInput(inputId, label, multiple, accept, width, buttonLabel, placeholder)

1

.....

- Choice A
- Choice B
- Choice C

Choice 1
Choice 1
Choice 2



Apply Changes

Enter text

numericInput(inputId, label, value, min, max, step, width)

passwordInput(inputId, label, value, width, placeholder)

radioButtons(inputId, label, choices, selected, inline, width, choiceNames, choiceValues)

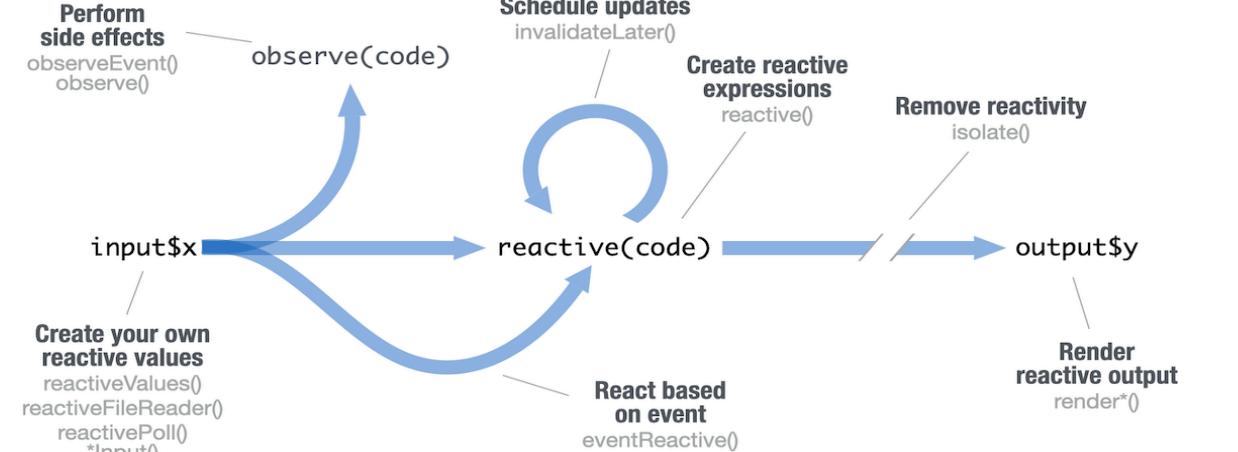
selectInput(inputId, label, choices, selected, multiple, selectize, width, size)
Also **selectizeInput()**

sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post, timeFormat, timezone, dragRange)

submitButton(text, icon, width)
(Prevent reactions for entire app)

textInput(inputId, label, value, width, placeholder) Also **textAreaInput()**

These are the core output types. See htmlwidgets.org for many more options.



CREATE YOUR OWN REACTIVE VALUES

```
# *Input() example
ui <- fluidPage(
 textInput("a","","A"))
```

```
#reactiveValues example
server <-
function(input,output){
  rv <- reactiveValues()
  rv$number <- 5
}
```

CREATE REACTIVE EXPRESSIONS

```
library(shiny)
ui <- fluidPage(
  textInput("a","","A"),
  textInput("z","","Z"),
  textOutput("b"))

server <-
function(input,output){
  re <- reactive({
    paste(input$a,input$z)
  })
  output$b <- renderText({
    re()
  })
}
shinyApp(ui, server)
```

REACT BASED ON EVENT

```
library(shiny)
ui <- fluidPage(
  textInput("a","","A"),
  actionButton("go","Go"),
  textOutput("b"))

server <-
function(input,output){
  re <- eventReactive(
    input$go,{input$a})
  output$b <- renderText({
    re()
  })
}
shinyApp(ui, server)
```

*Input() functions (see front page)

Each input function creates a reactive value stored as **input\$<inputId>**.

reactiveValues(...)
Creates a list of reactive values whose values you can set.

reactive(x, env, quoted, label, domain)

Reactive expressions:
• cache their value to reduce computation
• can be called elsewhere
• notify dependencies when invalidated
Call the expression with function syntax, e.g. **re()**.

RENDERS REACTIVE OUTPUT

```
library(shiny)
ui <- fluidPage(
  textInput("a","","A"),
  textOutput("b"))

server <-
function(input,output){
  output$b <-
  renderText({
    input$a
  })
}
shinyApp(ui, server)
```

render*() functions (see front page)

Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.

Save the results to **output\$<outputId>**.

PERFORM SIDE EFFECTS

```
library(shiny)
ui <- fluidPage(
  textInput("a","","A"),
  actionButton("go","Go"))

server <-
function(input,output){
  observeEvent(input$go,{
    print(input$a)
  })
}
shinyApp(ui, server)
```

observeEvent(eventExpr, handlerExpr, event.env, event.quoted, handler.env, handler.quoted, ..., label, suspended, priority, domain, autoDestroy, ignoreNULL, ignoreInit, once)
Runs code in 2nd argument when reactive values in 1st argument change. See **observe()** for alternative.

REMOVE REACTIVITY

```
library(shiny)
ui <- fluidPage(
  textInput("a","","A"),
  textOutput("b"))

server <-
function(input,output){
  output$b <-
  renderText({
    isolate({input$a})
  })
}
shinyApp(ui, server)
```

isolate(expr)

Runs a code block. Returns a **non-reactive** copy of the results.

```
fluidPage(
  textInput("a","", ""))
## <div class="container-fluid">
##   <div class="form-group shiny-input-container">
##     <label for="a"></label>
##     <input id="a" type="text"
##           class="form-control" value="" />
##   </div>
## </div>
```

Returns HTML

HTML Add static HTML elements with **tags**, a list of functions that parallel common HTML tags, e.g. **tags\$a()**. Unnamed arguments will be passed into the tag; named arguments will become tag attributes.

Run **names(tags)** for a complete list.
tags\$h1("Header") → <h1>Header</h1>

The most common tags have wrapper functions. You do not need to prefix their names with **tags\$**

```
ui <- fluidPage(
  h1("Header 1"),
  hr(),
  br(),
  p(strong("bold")),
  p(em("italic")),
  p(code("code")),
  a(href="", "link"),
  HTML("<p>Raw html</p>"))
)
```

Header 1

bold

italic

code

link

Raw html



To include a CSS file, use **includeCSS()**, or
1. Place the file in the **www** subdirectory
2. Link to it with:

```
tags$head(tags$link(rel = "stylesheet",
  type = "text/css", href = "<file name>"))
```



To include JavaScript, use **includeScript()** or
1. Place the file in the **www** subdirectory
2. Link to it with:

```
tags$head(tags$script(src = "<file name>"))
```

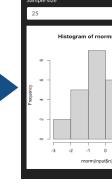


To include an image:
1. Place the file in the **www** subdirectory
2. Link to it with **img(src=<file name>)**

Themes

Use the **bslib** package to add existing themes to your Shiny app ui, or make your own.

```
library(bslib)
ui <- fluidPage(
  theme = bs_theme(
    bootswatch = "darkly",
    ...
  )
)
```



bootswatch_themes() Get a list of themes.

into a "single element" that has its own properties with a panel function, e.g.

```
wellPanel(
  dateInput("a", ""),
  submitButton()
)
```

Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

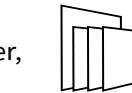
sidebarLayout()

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

fluidRow()

```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```

Also **flowLayout()**, **splitLayout()**, **verticalLayout()**, **fixedPage()**, and **fixedRow()**.



Layer tabPanels on top of each other, and navigate between them, with:

```
ui <- fluidPage( tabsetPanel(
  tabPanel("tab 1", "contents"),
  tabPanel("tab 2", "contents"),
  tabPanel("tab 3", "contents"))
)
```



```
ui <- fluidPage( navlistPanel(
  tabPanel("tab 1", "contents"),
  tabPanel("tab 2", "contents"),
  tabPanel("tab 3", "contents"))
)
```



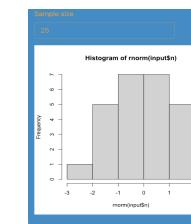
```
ui <- navbarPage(title = "Page",
  tabPanel("tab 1", "contents"),
  tabPanel("tab 2", "contents"),
  tabPanel("tab 3", "contents"))

```



Build your own theme by customizing individual arguments.

bs_theme(bg = "#558AC5", fg = "#F9B02D", ...)



?**bs_theme** for a full list of arguments.

bs_themer() Place within the server function to use the interactive theming widget.

Intro

sparklyr is an R interface for Apache Spark™.
It enables us to write all of our analysis code in R, but have the actual processing happen inside Spark clusters. Easily manipulate and model large-scale using R and Spark via **sparklyr**.



Import



Import data into Spark, not R

READ A FILE INTO SPARK

Arguments that apply to all functions:
sc, name, path, options=list(), repartition=0, memory=TRUE, overwrite=TRUE

CSV `spark_read_csv(header = TRUE, columns=NULL, infer_schema=TRUE, delimiter = "", quote = "", escape = "\\", charset = "UTF-8", null_value = NULL)`

JSON `spark_read_json()`

PARQUET `spark_read_parquet()`

TEXT `spark_read_text()`

ORC `spark_read_orc()`

LIBSVM `spark_read_libsvm()`

DELTA `spark_read_delta()`

AVRO `spark_read_avro()`

R DATA FRAME INTO SPARK

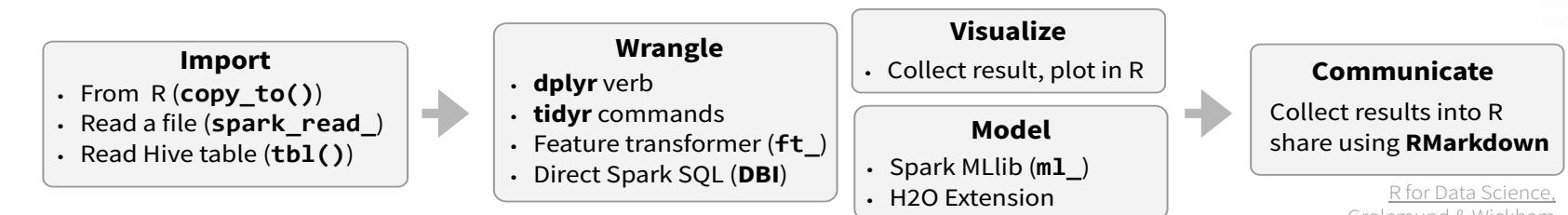
dplyr::`copy_to`(dest, df, name)

Apache Arrow accelerates data transfer between R and Spark. To use, simply load the library

ARROW ➡️ `library(sparklyr)`
`library(arrow)`

FROM A TABLE IN HIVE

HIVE dplyr::`tbl`(scr, ...) - Creates a reference to the table without loading it into memory



Wrangle

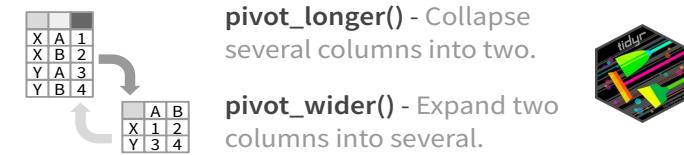
DPLYR VERBS

Translates into Spark SQL statements

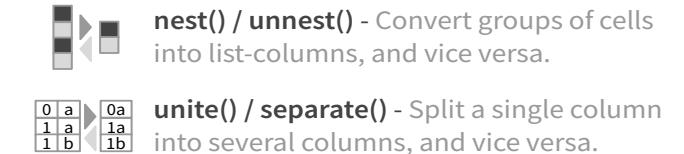
```
copy_to(sc, mtcars) %>%
  mutate(trm = ifelse(am == 0,
                      "auto", "man")) %>%
  group_by(trm) %>%
  summarise_all(mean)
```

TIDYR

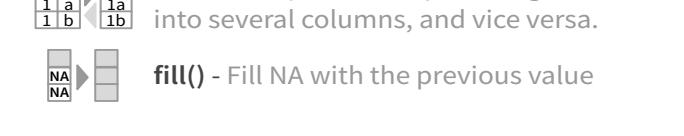
`pivot_longer()` - Collapse several columns into two.



`pivot_wider()` - Expand two columns into several.



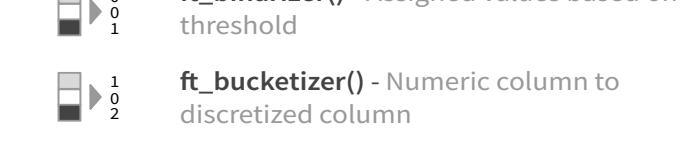
`nest()` / `unnest()` - Convert groups of cells into list-columns, and vice versa.



`unite()` / `separate()` - Split a single column into several columns, and vice versa.

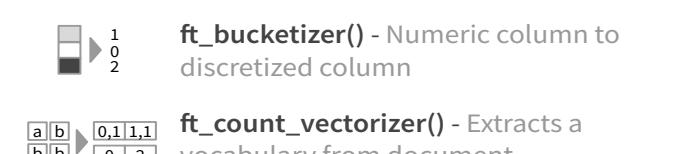


`fill()` - Fill NA with the previous value

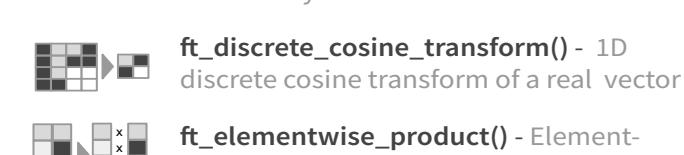


FEATURE TRANSFORMERS

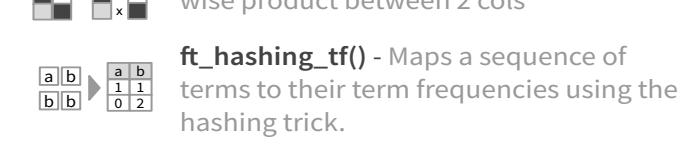
`ft_binarizer()` - Assigned values based on threshold



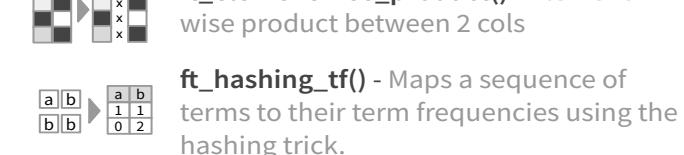
`ft_bucketizer()` - Numeric column to discretized column



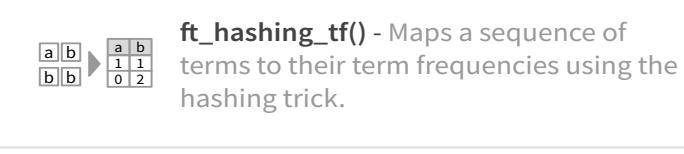
`ft_count_vectorizer()` - Extracts a vocabulary from document



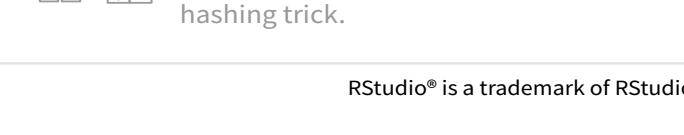
`ft_discrete_cosine_transform()` - 1D discrete cosine transform of a real vector



`ft_elementwise_product()` - Element-wise product between 2 cols



`ft_hashing_tf()` - Maps a sequence of terms to their term frequencies using the hashing trick.



`ft_idf()` - Compute the Inverse Document Frequency (IDF) given a collection of documents.

`ft_imputer()` - Imputation estimator for completing missing values, uses the mean or the median of the columns.

`ft_index_to_string()` - Index labels back to label as strings

`ft_interaction()` - Takes in Double and Vector columns and outputs a flattened vector of their feature interactions.

`ft_max_abs_scaler()` - Rescale each feature individually to range [-1, 1]

`ft_min_max_scaler()` - Rescale each feature to a common range [min, max] linearly

`ft_ngram()` - Converts the input array of strings into an array of n-grams

`ft_bucketed_random_projection_lsh()`

`ft_minhash_lsh()` - Locality Sensitive Hashing functions for Euclidean distance and Jaccard distance (MinHash)

`ft_normalizer()` - Normalize a vector to have unit norm using the given p-norm

`ft_one_hot_encoder()` - Continuous to binary vectors

`ft_pca()` - Project vectors to a lower dimensional space of top k principal components.

`ft_quantile_discretizer()` - Continuous to binned categorical values.

`ft_regex_tokenizer()` - Extracts tokens either by using the provided regex pattern to split the text.

`ft_robust_scaler()` - Removes the median and scales according to standard scale.

`ft_standard_scaler()` - Removes the mean and scaling to unit variance using column summary statistics

`ft_stop_words_remover()` - Filters out stop words from input

`ft_string_indexer()` - Column of labels into a column of label indices.

`ft_tokenizer()` - Converts to lowercase and then splits it by white spaces

`ft_vectorAssembler()` - Combine vectors into single row-vector

`ft_vector_indexer()` - Indexing categorical feature columns in a dataset of Vector

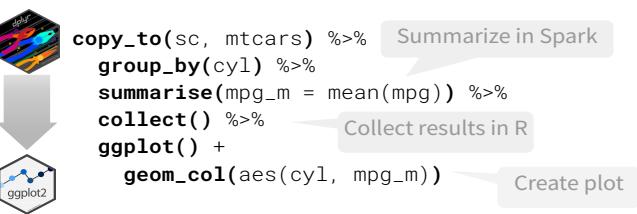
`ft_vector_slicer()` - Takes a feature vector and outputs a new feature vector with a subarray of the original features

`ft_word2vec()` - Word2Vec transforms a word into a code

Visualize



DPLYR + GGPLOT2



Modeling

REGRESSION

ml_linear_regression() - Linear regression.

ml_aft_survival_regression() - Parametric survival regression model named accelerated failure time (AFT) model

ml_generalized_linear_regression() - GLM

ml_isotonic_regression() - Currently implemented using parallelized pool adjacent violators algorithm. Only univariate (single feature) algorithm supported

ml_random_forest_regressor() - Regression using random forests.

CLASSIFICATION

ml_linear_svc() - Classification using linear support vector machines

ml_logistic_regression() - Logistic regression

ml_multilayer_perceptron_classifier() - Classification model based on the Multilayer Perceptron.

ml_naive_bayes() - It supports Multinomial NB which can handle finitely supported discrete data

ml_one_vs_rest() - Reduction of Multiclass Classification to Binary Classification. Performs reduction using one against all strategy.

TREE

ml_decision_tree_classifier() | **ml_decision_tree()** | **ml_decision_tree_regressor()** - Classification and regression using decision trees

ml_gbt_classifier() | **ml_gradient_boosted_trees()** | **ml_gbt_regressor()** - Binary classification and regression using gradient boosted trees

ml_random_forest_classifier() - Classification and regression using random forests.

ml_feature_importances() | **ml_tree_feature_importance()** - Feature Importance for Tree Models

CLUSTERING

ml_bisecting_kmeans() - A bisecting k-means algorithm based on the paper

ml_lda() | **ml_describe_topics()** | **ml_log_likelihood()** | **ml_log_perplexity()** | **ml_topics_matrix()** - LDA topic model designed for text documents.

ml_gaussian_mixture() - Expectation maximization for multivariate Gaussian Mixture Models (GMMs)

ml_kmeans() | **ml_compute_cost()** | **ml_compute_silhouette_measure()** - Clustering with support for k-means

ml_power_iteration() - For clustering vertices of a graph given pairwise similarities as edge properties.

FEATURE

ml_chisquare_test(x,features,label) - Pearson's independence test for every feature against the label

ml_default_stop_words() - Loads the default stop words for the given language

STATS

ml_summary() - Extracts a metric from the summary object of a Spark ML model

ml_corr() - Compute correlation matrix

RECOMMENDATION

ml_als() | **ml_recommend()** - Recommendation using Alternating Least Squares matrix factorization

EVALUATION

ml_clustering_evaluator() - Evaluator for clustering

ml_evaluate() - Compute performance metrics

ml_binary_classification_evaluator() | **ml_binary_classification_eval()** | **ml_classification_eval()** - A set of functions to calculate performance metrics for prediction models.

FREQUENT PATTERN

ml_fpgrowth() | **ml_association_rules()** | **ml_freq_itemsets()** - A parallel FP-growth algorithm to mine frequent itemsets.

ml_freq_seq_patterns() | **ml_prefixspan()** - PrefixSpan algorithm for mining frequent itemsets.

UTILITIES

ml_call_constructor() - Identifies the associated sparklyr ML constructor for the JVM

ml_model_data() - Extracts data associated with a Spark ML model

ml_standardize_formula() - Generates a formula string from user inputs, to be used in `ml_model` constructor

ml_uid() - Extracts the UID of an ML object.

ML Pipelines

Easily create a formal Spark Pipeline models using R. Save the Pipeline in native Scala. The saved model will have no dependencies on R.

INITIALIZE AND TRAIN

ml_pipeline() - Initializes a new Spark Pipeline

ml_fit() - Trains the model, outputs a Spark Pipeline Model.

SAVE AND RETRIEVE

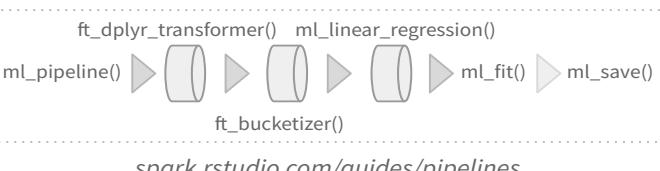
ml_save() - Saves into a format that can be read by Scala and PySpark .

ml_read() - Reads Spark object into sparklyr.

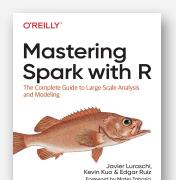
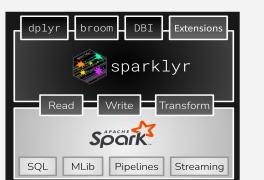
SQL AND DPLYR

ft_sql_transformer() - Creates a Pipeline step based on the SQL statement passed to the command.

ft_dplyr_transformer() - Creates a Pipeline step based on one or several dplyr commands.



More Info



spark.rstudio.com

therinspark.com

Sessions



YARN CLIENT

1. Install RStudio Server on an edge node
2. Locate path to the cluster's Spark Home Directory, it normally is `"/usr/lib/spark"`
3. Basic configuration example

```
conf <- spark_config()
conf$spark.executor.memory <- "300M"
conf$spark.executor.cores <- 2
conf$spark.executor.instances <- 3
conf$spark.dynamicAllocation.enabled<-"false"
```
4. Open a connection

```
sc <- spark_connect(master = "yarn",
                      spark_home = "/usr/lib/spark/",
                      version = "2.1.0", config = conf)
```

YARN CLUSTER

1. Make sure to have copies of the `yarn-site.xml` and `hive-site.xml` files in the RStudio Server
2. Point environment variables to the correct paths

```
Sys.setenv(JAVA_HOME="[Path]")
Sys.setenv(SPARK_HOME ="[Path]")
Sys.setenv(YARN_CONF_DIR ="[Path]")
```
3. Open a connection

```
sc <- spark_connect(master = "yarn-cluster")
```

STANDALONE CLUSTER

1. Install RStudio Server on one of the existing nodes or a server in the same LAN
2. Open a connection

```
spark_connect(master="spark://host:port",
              version = "2.0.1",
              spark_home = [path to Spark])
```

LOCAL MODE

No cluster required. Use for learning purposes only

1. Install a local version of Spark: `spark_install()`
2. Open a connection

```
sc <- spark_connect(master="local")
```

KUBERNETES

1. Use the following to obtain the Host and Port

```
system2("kubectl", "cluster-info")
```
2. Open a connection

```
sc <- spark_connect(config =
                     spark_config_kubernetes(
                     "k8s://https://[HOST]:[PORT]",
                     account = "default",
                     image = "docker.io/owner/repo:version"))
```

CLOUD

- Databricks** - `spark_connect(method = "databricks")`
- Qubole**- `spark_connect(method = "qubole")`



The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

Detect Matches



str_detect(string, pattern, negate = FALSE)
Detect the presence of a pattern match in a string. Also **str_like()**. `str_detect(fruit, "a")`



str_starts(string, pattern, negate = FALSE)
Detect the presence of a pattern match at the beginning of a string. Also **str_ends()**. `str_starts(fruit, "a")`



str_which(string, pattern, negate = FALSE)
Find the indexes of strings that contain a pattern match. `str_which(fruit, "a")`



str_locate(string, pattern) Locate the positions of pattern matches in a string. Also **str_locate_all()**. `str_locate(fruit, "a")`



str_count(string, pattern) Count the number of matches in a string. `str_count(fruit, "a")`

Mutate Strings



str_sub() <- value. Replace substrings by identifying the substrings with `str_sub()` and assigning into the results. `str_sub(fruit, 1, 3) <- "str"`



str_replace(string, pattern, replacement) Replace the first matched pattern in each string. Also **str_remove()**. `str_replace(fruit, "p", "-")`



str_replace_all(string, pattern, replacement) Replace all matched patterns in each string. Also **str_remove_all()**. `str_replace_all(fruit, "p", "-")`

A STRING
↓
a string

str_to_lower(string, locale = "en")¹
Convert strings to lower case. `str_to_lower(sentences)`

a string
↓
A STRING

str_to_upper(string, locale = "en")¹
Convert strings to upper case. `str_to_upper(sentences)`

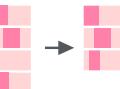
a string
↓
A String

str_to_title(string, locale = "en")¹ Convert strings to title case. Also **str_to_sentence()**. `str_to_title(sentences)`

Subset Strings



str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector. `str_sub(fruit, 1, 3); str_sub(fruit, -2)`



str_subset(string, pattern, negate = FALSE)
Return only the strings that contain a pattern match. `str_subset(fruit, "p")`



str_extract(string, pattern) Return the first pattern match found in each string, as a vector. Also **str_extract_all()** to return every pattern match. `str_extract(fruit, "[aeiou]")`



str_match(string, pattern) Return the first pattern match found in each string, as a matrix with a column for each () group in pattern. Also **str_match_all()**. `str_match(sentences, "(a|the) ([^]+)")`

Manage Lengths



str_length(string) The width of strings (i.e. number of code points, which generally equals the number of characters). `str_length(fruit)`



str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. `str_pad(fruit, 17)`



str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis. `str_trunc(sentences, 6)`



str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. `str_trim(str_pad(fruit, 17))`



str_squish(string) Trim whitespace from each end and collapse multiple spaces into single spaces. `str_squish(str_pad(fruit, 17, "both"))`

Join and Split



str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string. `str_c(letters, LETTERS)`



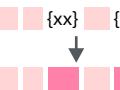
str_flatten(string, collapse = "") Combines into a single string, separated by collapse. `str_flatten(fruit, ", ")`



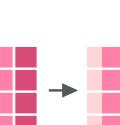
str_dup(string, times) Repeat strings times times. Also **str_unique()** to remove duplicates. `str_dup(fruit, times = 2)`



str_split_fixed(string, pattern, n) Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split()** to return a list of substrings and **str_split_n()** to return the nth substring. `str_split_fixed(sentences, " ", n=3)`



str_glue(..., .sep = "", .envir = parent.frame()) Create a string from strings and {expressions} to evaluate. `str_glue("Pi is {pi}")`



str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA") Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate. `str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")`

Order Strings



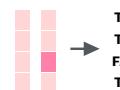
str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)¹ Return the vector of indexes that sorts a character vector. `fruit[str_order(fruit)]`



str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)¹ Sort a character vector. `str_sort(fruit)`

Helpers

appl<e>
banana
p<e>ar



This is a long sentence.
↓
This is a long sentence.

str_conv(string, encoding) Override the encoding of a string. `str_conv(fruit, "ISO-8859-1")`

str_view_all(string, pattern, match = NA) View HTML rendering of all regex matches. Also **str_view()** to see only the first match. `str_view_all(sentences, "[aeiou])`

str_equal(x, y, locale = "en", ignore_case = FALSE, ...)¹ Determine if two strings are equivalent. `str_equal(c("a", "b"), c("a", "c"))`

str_wrap(string, width = 80, indent = 0, exdent = 0) Wrap strings into nicely formatted paragraphs. `str_wrap(sentences, 20)`

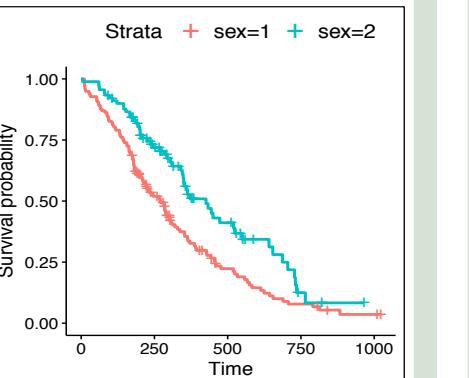
¹ See bit.ly/ISO639-1 for a complete list of locales.

Survival Curves with survminer

Survival Curves

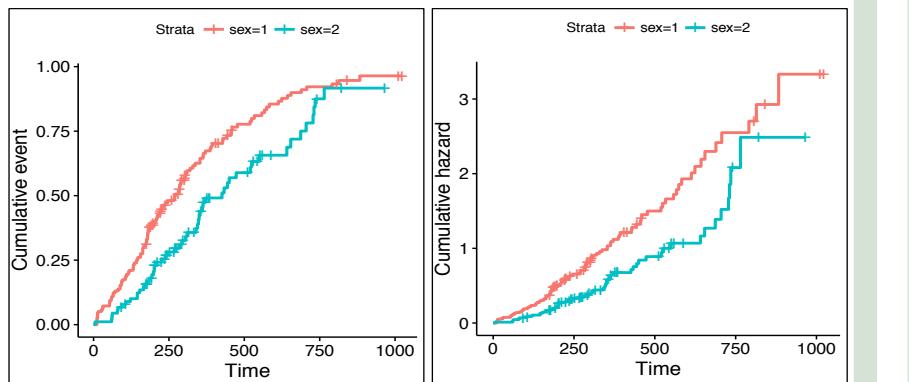
The **ggsurvplot()** function creates **ggplot2** plots from **survfit** objects.

```
library("survival")
fit <- survfit(Surv(time, status) ~ sex, data = lung)
class(fit)
## [1] "survfit"
library("survminer")
ggsurvplot(fit, data = lung)
```



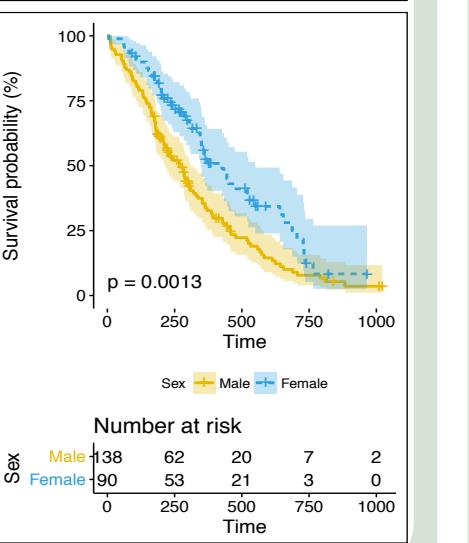
Use the **fun** argument to set the transformation of the survival curve. E.g. "**event**" for cumulative events, "**cumhaz**" for the cumulative hazard function or "**pct**" for survival probability in percentage.

```
ggsurvplot(fit, data = lung, fun = "event")
ggsurvplot(fit, data = lung, fun = "cumhaz")
```



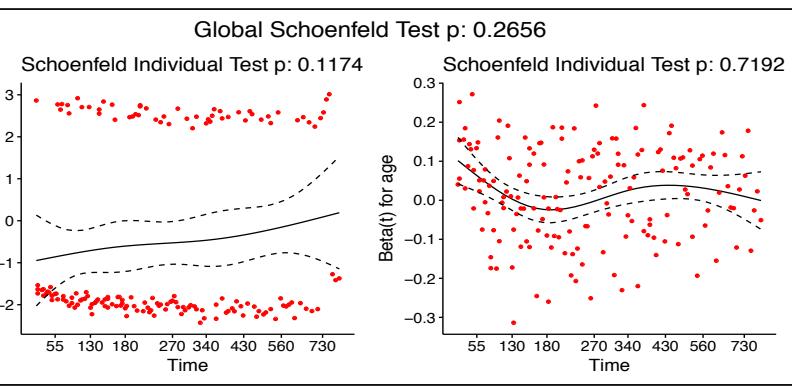
With lots of graphical parameters you have full control over look and feel of the survival plots; position and content of the legend; additional annotations like p-value, title, subtitle.

```
ggsurvplot(fit, data = lung,
conf.int = TRUE,
pval = TRUE,
fun = "pct",
risk.table = TRUE,
size = 1,
linetype = "strata",
palette = c("#E7B800",
 "#2E9FDF"),
legend = "bottom",
legend.title = "Sex",
legend.labs = c("Male",
 "Female"))
```



proportional hazards assumption for a Cox regression model fit. The graphical verification of this assumption may be performed with the function **ggcoxph()** from the **survminer** package. For each covariate it produces plots with scaled Schoenfeld residuals against the time.

```
library("survival")
fit <- coxph(Surv(time, status) ~ sex + age, data = lung)
ftest <- cox.zph(fit)
ftest
##          rho chisq      p
## sex     0.1236 2.452 0.117
## age    -0.0275 0.129 0.719
## GLOBAL    NA 2.651 0.266
library("survminer")
ggcoxph(ftest)
```



The function **ggcoxdiagnostics()** plots different types of residuals as a function of time, linear predictor or observation id. The type of residual is selected with **type** argument. Possible values are "martingale", "deviance", "score", "schoenfeld", "dfbeta", "dfbetas", and "scaledsch".

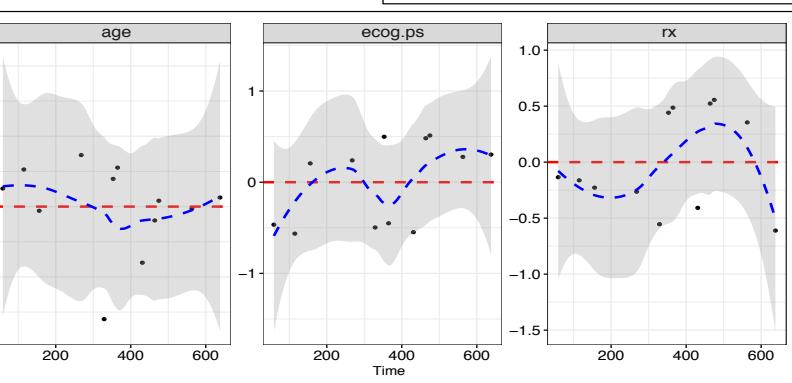
The **ox.scale** argument defines what shall be plotted on the OX axis. Possible values are "linear.predictions", "observation.id", "time".

Logical arguments **hline** and **sline** may be used to add horizontal line or smooth line to the plot.

```
library("survival")
library("survminer")
fit <- coxph(Surv(time, status) ~ sex + age, data = lung)
```

```
ggcoxdiagnostics(fit,
type = "deviance",
ox.scale = "linear.predictions")
```

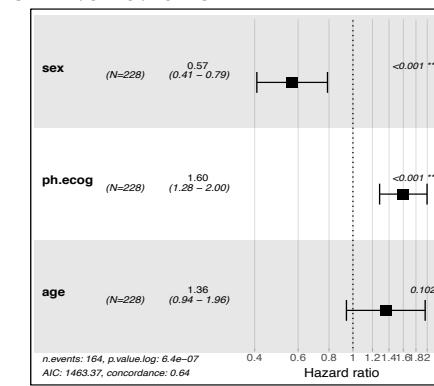
```
ggcoxdiagnostics(fit,
type = "schoenfeld",
ox.scale = "time")
```



for a Cox regression model fit. Hazard ratio estimates along with confidence intervals and p-values are plotted for each variable.

```
library("survival")
library("survminer")
lung$age <- ifelse(lung$age > 70, ">70", "<= 70")
fit <- coxph( Surv(time, status) ~ sex + ph.ecog + age, data = lung)
fit

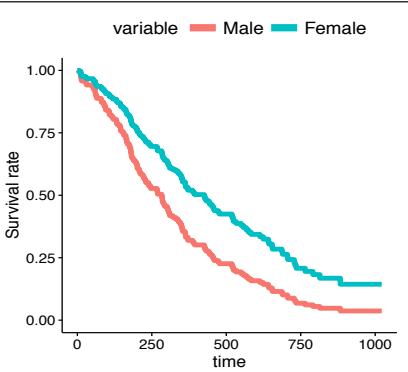
## Call:
## coxph(formula = Surv(time, status) ~ sex+ph.ecog+age, data=lung)
##
##            coef exp(coef) se(coef)      z      p
## sex       -0.567    0.567   0.168 -3.37 0.00075
## ph.ecog    0.470    1.600   0.113  4.16 3.1e-05
## age>70    0.307    1.359   0.187  1.64 0.10175
##
## Likelihood ratio test=31.6 on
## n= 227, number of events= 164
ggforest(fit)
```



The function **ggadjustedcurves()** from the **survminer** package plots Adjusted Survival Curves for Cox Proportional Hazards Model. Adjusted Survival Curves show how a selected factor influences survival estimated from a Cox model.

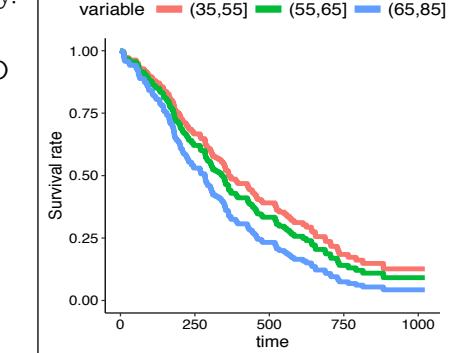
Note that these curves differ from Kaplan Meier estimates since they present expected survival based on given Cox model.

```
library("survival")
library("survminer")
lung$sex <- ifelse(lung$sex == 1,
"Male", "Female")
fit <- coxph(Surv(time, status) ~ sex + ph.ecog + age +
strata(sex),
data = lung)
ggadjustedcurves(fit, data=lung)
```



Note that it is not necessary to include the grouping factor in the Cox model. Survival curves are estimated from Cox model for each group defined by the factor independently.

```
lung$age3 <- cut(lung$age,
c(35,55,65,85))
ggadjustedcurves(fit, data=lung,
variable="age3")
```



Dollar sign syntax

```
goal(data$x, data$y)
```

SUMMARY STATISTICS:

one continuous variable:

```
mean(mtcars$mpg)
```

one categorical variable:

```
table(mtcars$cyl)
```

two categorical variables:

```
table(mtcars$cyl, mtcars$am)
```

one continuous, one categorical:

```
mean(mtcars$mpg [mtcars$cyl==4])
```

```
mean(mtcars$mpg [mtcars$cyl==6])
```

```
mean(mtcars$mpg [mtcars$cyl==8])
```

PLOTTING:

one continuous variable:

```
hist(mtcars$disp)
```

```
boxplot(mtcars$disp)
```

one categorical variable:

```
barplot(table(mtcars$cyl))
```

two continuous variables:

```
plot(mtcars$disp, mtcars$mpg)
```

two categorical variables:

```
mosaicplot(table(mtcars$am, mtcars$cyl))
```

one continuous, one categorical:

```
histogram(mtcars$disp[mtcars$cyl==4])
```

```
histogram(mtcars$disp[mtcars$cyl==6])
```

```
histogram(mtcars$disp[mtcars$cyl==8])
```

```
boxplot(mtcars$disp[mtcars$cyl==4])
```

```
boxplot(mtcars$disp[mtcars$cyl==6])
```

```
boxplot(mtcars$disp[mtcars$cyl==8])
```

WRANGLING:

subsetting:

```
mtcars [mtcars$mpg>30, ]
```

making a new variable:

```
mtcars$efficient [mtcars$mpg>30] <- TRUE
```

```
mtcars$efficient [mtcars$mpg<30] <- FALSE
```

Formula syntax

```
goal(y~x|z, data=data, group=w)
```

SUMMARY STATISTICS:

one continuous variable:

```
mosaic::mean(~mpg, data=mtcars)
```

one categorical variable:

```
mosaic::tally(~cyl, data=mtcars)
```

two categorical variables:

```
mosaic::tally(cyl~am, data=mtcars)
```

one continuous, one categorical:

```
mosaic::mean(mpg~cyl, data=mtcars)
```

tilde

PLOTTING:

one continuous variable:

```
lattice::histogram(~disp, data=mtcars)
```

```
lattice::bwplot(~disp, data=mtcars)
```

one categorical variable:

```
mosaic::bargraph(~cyl, data=mtcars)
```

two continuous variables:

```
lattice::xyplot(mpg~disp, data=mtcars)
```

two categorical variables:

```
mosaic::bargraph(~am, data=mtcars, group=cyl)
```

one continuous, one categorical:

```
lattice::histogram(~disp|cyl, data=mtcars)
```

```
lattice::bwplot(cyl~disp, data=mtcars)
```

The variety of R syntaxes give you many ways to “say” the same thing

read across the cheatsheet to see how different syntaxes approach the same problem

Tidyverse syntax

```
data %>% goal(x)
```

SUMMARY STATISTICS:

one continuous variable:

```
mtcars %>% dplyr::summarize(mean(mpg))
```

one categorical variable:

```
mtcars %>% dplyr::group_by(cyl) %>%  
dplyr::summarize(n())
```

two categorical variables:

```
mtcars %>% dplyr::group_by(cyl, am) %>%  
dplyr::summarize(n())
```

one continuous, one categorical:

```
mtcars %>% dplyr::group_by(cyl) %>%  
dplyr::summarize(mean(mpg))
```

the pipe

PLOTTING:

one continuous variable:

```
ggplot2::qplot(x=mpg, data=mtcars, geom = "histogram")
```

```
ggplot2::qplot(y=disp, x=1, data=mtcars, geom="boxplot")
```

one categorical variable:

```
ggplot2::qplot(x=cyl, data=mtcars, geom="bar")
```

two continuous variables:

```
ggplot2::qplot(x=disp, y=mpg, data=mtcars, geom="point")
```

two categorical variables:

```
ggplot2::qplot(x=factor(cyl), data=mtcars, geom="bar") +  
facet_grid(.~am)
```

one continuous, one categorical:

```
ggplot2::qplot(x=disp, data=mtcars, geom = "histogram") +  
facet_grid(.~cyl)
```

```
ggplot2::qplot(y=disp, x=factor(cyl), data=mtcars,  
geom="boxplot")
```

WRANGLING:

subsetting:

```
mtcars %>% dplyr::filter(mpg>30)
```

making a new variable:

```
mtcars <- mtcars %>%  
dplyr::mutate(efficient = if_else(mpg>30, TRUE, FALSE))
```

Syntax is the set of rules that govern what code works and doesn't work in a programming language. Most programming languages offer one standardized syntax, but R allows package developers to specify their own syntax. As a result, there is a large variety of (equally valid) R syntaxes.

The three most prevalent R syntaxes are:

1. The **dollar sign syntax**, sometimes called **base R syntax**, expected by most base R functions. It is characterized by the use of `dataset$variableName`, and is also associated with square bracket subsetting, as in `dataset[1, 2]`. Almost all R functions will accept things passed to them in dollar sign syntax.
2. The **formula syntax**, used by modeling functions like `lm()`, lattice graphics, and mosaic summary statistics. It uses the tilde (~) to connect a response variable and one (or many) predictors. Many base R functions will accept formula syntax.
3. The **tidyverse syntax** used by `dplyr`, `tidyR`, and more. These functions expect data to be the first argument, which allows them to work with the "pipe" (%>%) from the `magrittr` package. Typically, `ggplot2` is thought of as part of the tidyverse, although it has its own flavor of the syntax using plus signs (+) to string pieces together. `ggplot2` author Hadley Wickham has said the package would have had different syntax if he had written it after learning about the pipe.

Educators often try to teach within one unified syntax, but most R programmers use some combination of all the syntaxes.

Internet research tip:

If you are searching on google, StackOverflow, or another favorite online source and see code in a syntax you don't recognize:

- Check to see if the code is using one of the three common syntaxes listed on this cheatsheet
- Try your search again, using a keyword from the syntax name ("tidyverse") or a relevant package ("mosaic")



Sometimes particular syntaxes work, but are considered dangerous to use, because they are so easy to get wrong. For example, passing variable names without assigning them to a named argument.

Even more ways to say the same thing

Even within one syntax, there are often variations that are equally valid. As a case study, let's look at the `ggplot2` syntax. `ggplot2` is the plotting package that lives within the tidyverse. If you read `down` this column, all the code here produces the same graphic.

quickplot

`qplot()` stands for quickplot, and allows you to make quick plots. It doesn't have the full power of `ggplot2`, and it uses a slightly different syntax than the rest of the package.

```
ggplot2::qplot(x=disp, y=mpg, data=mtcars, geom="point")
```

```
ggplot2::qplot(x=disp, y=mpg, data=mtcars) ⓘ
```

```
ggplot2::qplot(disp, mpg, data=mtcars) ⓘ ⓘ
```

read down this column for many pieces of code in one syntax that look different but produce the same graphic

ggplot

To unlock the power of `ggplot2`, you need to use the `ggplot()` function (which sets up a plotting region) and add geoms to the plot.

```
ggplot2::ggplot(mtcars) +  
  geom_point(aes(x=disp, y=mpg))
```

```
ggplot2::ggplot(data=mtcars) +  
  geom_point(mapping=aes(x=disp, y=mpg))
```

plus adds layers

```
ggplot2::ggplot(mtcars, aes(x=disp, y=mpg)) +  
  geom_point()
```

```
ggplot2::ggplot(mtcars, aes(x=disp)) +  
  geom_point(aes(y=mpg))
```

ggformula

The "third and a half way" to use the formula syntax, but get `ggplot2`-style graphics

```
ggformula::gf_point(mpg~disp, data= mtcars)
```

formulas in base plots

Base R plots will also take the formula syntax, although it's not as commonly used

```
plot(mpg~disp, data=mtcars)
```



Before the course (design)

Use these to prepare your lecture/course:

Who are your learners? (Persona Analysis)
(change according to requirements...[1])



The R novice

Background: some statistics, some programming

Prior knowledge: basic R course, base R syntax

Goals: understand tidy concepts,
expose to tidyverse practices

Special needs: First successes, mitigate fears, encourage learning

The R “false expert”

Background: working with R for some time, but doesn't keep-up

Prior knowledge: been using base R syntax, loops, and functions

Goals: strengthen tidyverse familiarity, apply dplyr workflow

Special needs: switch from obsolete methods to state-of-the-art R

Define goals using **Bloom's Taxonomy** [2]

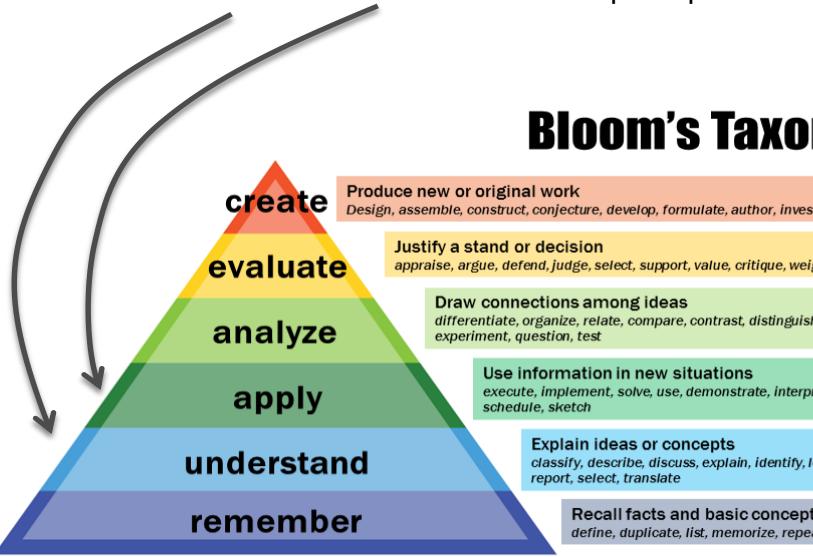
Design your classes to move your learners “up the pyramid”



Keep “realistic goals” for each persona

For example (R shiny - novice):

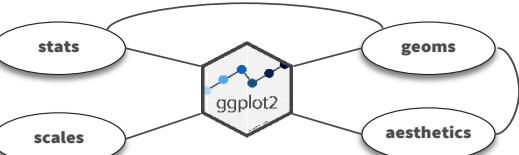
Learners will **describe** and **demonstrate** the principles of reactivity



Bloom's Taxonomy

Design your lecture using **Conceptual maps**

Keep the number of elements small (up to ~7 items), e.g.:



Write the “final exam”

How are you going to test knowledge after the lecture?

What should learners be able to answer?

Turn the concepts into slides



Add faded examples (exercises) and check-in slides

Check-ins, e.g.: multiple choice quick questions”

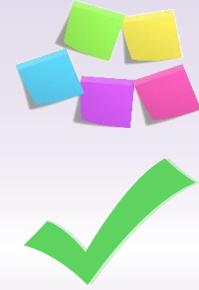
Faded examples = fill in the blanks, e.g.:

`ggplot(data = ___, mapping = aes(x = ___, y = ___)) +
 geom_*(___) +`

During the course (implement)

Things you can implement to improve the lecture workflow

Learners use **sticky notes** to indicate status:



Green = everything is OK/Exercise completed.

Red = help me! I ran into a problem.

Blue = I need a break.

Engage with online mini-polls during lecture
Use regular check-ins during your lecture (3-4 check-ins per hour). No one “opts-out” (everyone answers once in a while – you choose)

Additional sources

[1] Dreyfus, Stuart E., and Hubert L. Dreyfus. *A five stage model of the mental activities involved in directed skill acquisition*. No. ORC-80-2. California Univ Berkeley Operations Research Center, 1980.

[2] Content downloaded from <https://cft.vanderbilt.edu/guides-sub-pages/blooms-taxonomy/>
(CC-BY-SA Vanderbilt University Center for Teaching)

After the course (learn/improve)

Make sure you make the most to improve your next lecture



Use feedback to understand what went well, and what you need to improve.



Measure the time each lecture takes you (or where did you get to), so that next time your time estimates will be better

Useful tips and tricks

Useful tips for preparations



Use github to upload course materials



RMarkdown for exercises

Recommended reading materials/references for R courses:

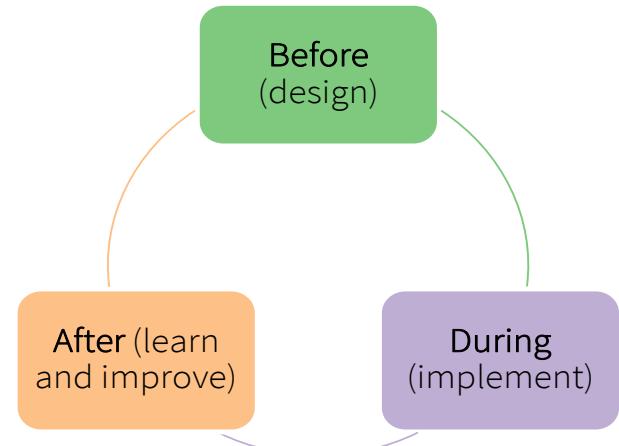
R for Data Science / Garrett Grolemund and Hadley Wickham (r4ds.had.co.nz)

Advanced R / Hadley Wickham (adv-r.had.co.nz)

RStudio **Cheat sheets**:

<https://www.rstudio.com/resources/cheatsheets/>

Iterative work flow



Tidy data is a way to organize tabular data in a consistent data structure across packages.

A table is tidy if:

A	B	C
↑	↑	↑
↓	↓	↓

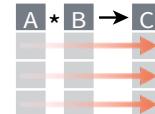
&

A	B	C
↔	↔	↔
↔	↔	↔

Each **variable** is in its own **column**

Each **observation**, or **case**, is in its own row

A	B	C
Yellow	Orange	Green
Yellow	Orange	Green



Access **variables** as **vectors**

Preserve **cases** in vectorized operations

Tibbles

AN ENHANCED DATA FRAME

Tibbles are a table format provided by the **tibble** package. They inherit the data frame class, but have improved behaviors:

- **Subset** a new tibble with `[]`, a vector with `[[]]` and `$`.
- **No partial matching** when subsetting columns.
- **Display** concise views of the data on one screen.

`options(tibble.print_max = n, tibble.print_min = m, tibble.width = Inf)` Control default display settings.

`View()` or `glimpse()` View the entire data set.

CONSTRUCT A TIBBLE

`tibble(...)` Construct by columns.

`tibble(x = 1:3, y = c("a", "b", "c"))`

Both make this tibble

A tibble: 3 × 2
 x y
 <int> <chr>
1 1 a
2 2 b
3 3 c

`as_tibble(x, ...)` Convert a data frame to a tibble.

`enframe(x, name = "name", value = "value")`

Convert a named vector to a tibble. Also `deframe()`.

`is_tibble(x)` Test whether x is a tibble.



Reshape Data

- Pivot data to reorganize values into a new layout.

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K



country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T



country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

Split Cells

- Use these functions to split or combine cells into individual, isolated values.

table5

country	century	year
A	19	99
A	20	00
B	19	99
B	20	00



country	year
A	1999
A	2000
B	1999
B	2000

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M



country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M



country	year	rate
A	1999	0.7K
A	1999	19M
A	2000	2K
A	2000	20M
B	1999	37K
B	1999	172M
B	2000	80K
B	2000	174M

pivot_longer(data, cols, names_to = "name", values_to = "value", values_drop_na = FALSE)

"Lengthen" data by collapsing several columns into two. Column names move to a new names_to column and values to a new values_to column.

```
pivot_longer(table4a, cols = 2:3, names_to = "year", values_to = "cases")
```

pivot_wider(data, names_from = "name", values_from = "value")

The inverse of pivot_longer(). "Widen" data by expanding two columns into several. One column provides the new column names, the other the values.

```
pivot_wider(table2, names_from = type, values_from = count)
```

Expand Tables

Create new combinations of variables or identify implicit missing values (combinations of variables not present in the data).

x	x1	x2	x3
A	1	3	
B	1	4	
B	2	3	

expand(data, ...) Create a new tibble with all possible combinations of the values of the variables listed in ...

Drop other variables.

```
expand(mtcars, cyl, gear, carb)
```

x	x1	x2	x3
A	1	3	
B	1	4	
B	2	3	

complete(data, ..., fill = list()) Add missing possible combinations of values of variables listed in ... Fill remaining variables with NA.

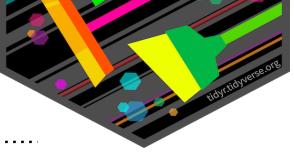
```
complete(mtcars, cyl, gear, carb)
```

x	x1	x2
A	1	
B	NA	
C	NA	
D	3	
E	NA	

drop_na(data, ...) Drop rows containing NA's in ... columns.

```
drop_na(x, x2)
```

x	x1	x2
A	1	
B	2	



Use a nested data frame to:

- Preserve relationships between observations and subsets of data. Preserve the type of the variables being nested (factors and datetimes aren't coerced to character).
- Manipulate many sub-tables at once with **purrr** functions like `map()`, `map2()`, or `pmap()` or with **dplyr** `rowwise()` grouping.

CREATE NESTED DATA

nest(data, ...) Moves groups of cells into a list-column of a data frame. Use alone or with `dplyr::group_by()`:

1. Group the data frame with `group_by()` and use `nest()` to move the groups into a list-column.

```
n_storms <- storms %>%
  group_by(name) %>%
  nest()
```

2. Use `nest(new_col = c(x, y))` to specify the columns to group using `dplyr::select()` syntax.

```
n_storms <- storms %>%
  nest(data = c(year:long))
```

name				yr	lat	long	name				yr	lat	long	
Amy	1975	27.5	-79.0	Amy	1975	27.5	-79.0							
Amy	1975	28.5	-79.0	Amy	1975	28.5	-79.0							
Amy	1975	29.5	-79.0	Amy	1975	29.5	-79.0							
Bob	1979	22.0	-96.0	Bob	1979	22.0	-96.0							
Bob	1979	22.5	-95.3	Bob	1979	22.5	-95.3							
Bob	1979	23.0	-94.6	Bob	1979	23.0	-94.6							
Zeta	2005	23.9	-35.6	Zeta	2005	23.9	-35.6							
Zeta	2005	24.2	-36.1	Zeta	2005	24.2	-36.1							
Zeta	2005	24.7	-36.6	Zeta	2005	24.7	-36.6							

Index list-columns with `[[[]]]`. `n_storms$data[[1]]`

CREATE TIBBLES WITH LIST-COLUMNS

tibble::tribble(...) Makes list-columns when needed.

```
tibble(~max, ~seq,
      3, 1:3,
      4, 1:4,
      5, 1:5)
```

max	seq
3	<int [3]>
4	<int [4]>
5	<int [5]>

tibble::tibble(...) Saves list input as list-columns.

```
tibble(max = c(3, 4, 5), seq = list(1:3, 1:4, 1:5))
```

tibble::enframe(x, name="name", value="value")

Converts multi-level list to a tibble with list-cols.
`enframe(list('3'=1:3, '4'=1:4, '5'=1:5), 'max', 'seq')`

OUTPUT LIST-COLUMNS FROM OTHER FUNCTIONS

dplyr::mutate(), transmute(), and summarise() will output list-columns if they return a list.

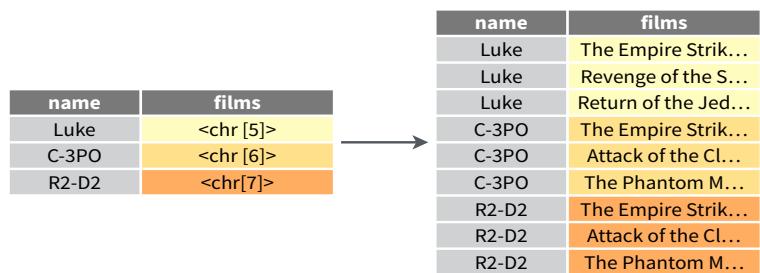
```
mtcars %>%
  group_by(cyl) %>%
  summarise(q = list(quantile(mpg)))
```

RESHAPE NESTED DATA

unnest(data, cols, ..., keep_empty = FALSE) Flatten nested columns back to regular columns. The inverse of `nest()`.
`n_storms %>% unnest(data)`

unnest_longer(data, col, values_to = NULL, indices_to = NULL)
Turn each element of a list-column into a row.

```
starwars %>%
  select(name, films) %>%
  unnest_longer(films)
```



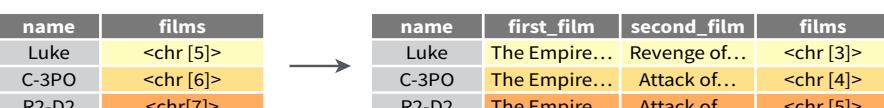
unnest_wider(data, col) Turn each element of a list-column into a regular column.

```
starwars %>%
  select(name, films) %>%
  unnest_wider(films)
```



hoist(.data, .col, ..., .remove = TRUE) Selectively pull list components out into their own top-level columns. Uses `purrr::pluck()` syntax for selecting from lists.

```
starwars %>%
  select(name, films) %>%
  hoist(films, first_film = 1, second_film = 2)
```



TRANSFORM NESTED DATA

A vectorized function takes a vector, transforms each element in parallel, and returns a vector of the same length. By themselves vectorized functions cannot work with lists, such as list-columns.

dplyr::rowwise(.data, ...) Group data so that each row is one group, and within the groups, elements of list-columns appear directly (accessed with `[[]]`, not as lists of length one. When you use `rowwise()`, **dplyr** functions will seem to apply functions to list-columns in a vectorized fashion.



Apply a function to a list-column and **create a new list-column**.

`n_storms %>% rowwise() %>% mutate(n = list(dim(data)))`

dim() returns two values per row
wrap with list to tell mutate to create a list-column

Apply a function to a list-column and **create a regular column**.

`n_storms %>% rowwise() %>% mutate(n = nrow(data))`

nrow() returns one integer per row

Collapse multiple list-columns into a single list-column.

`starwars %>% rowwise() %>% mutate(transport = list(append(vehicles, starships)))`

append() returns a list for each row, so col type must be list

Apply a function to **multiple list-columns**.

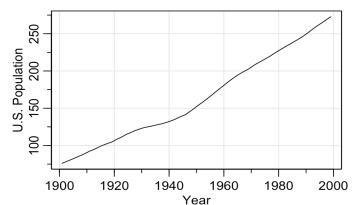
`starwars %>% rowwise() %>% mutate(n_transports = length(c(vehicles, starships)))`

length() returns one integer per row

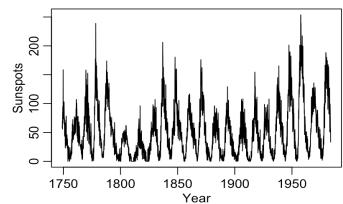
See **purrr** package for more list functions.

Plot Time Series

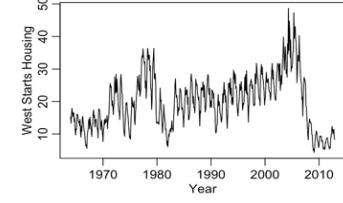
1. tsplot(x=time, y=data)



2. plot(ts(data, start=start_time, frequency=gap))



3. ts.plot(ts(data, start=start_time, frequency=gap))



Simulation

Autoregression of Order p

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \dots + \phi_p X_{t-p} + W_t$$

Moving Average of Order q

$$Z_t = Z_t + \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \dots + \theta_q Z_{t-q}$$

ARMA (p, q)

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \dots + \phi_p X_{t-p} + Z_t + \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \dots + \theta_q Z_{t-q}$$

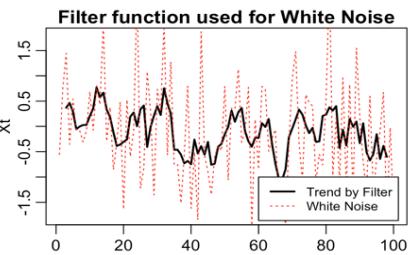
Simulation of ARMA (p, q)

```
arima.sim(model=list(ar=c(phi1, ..., phi_p),
                     ma=c(theta1, ..., theta_q)), n=n)
```

Filters

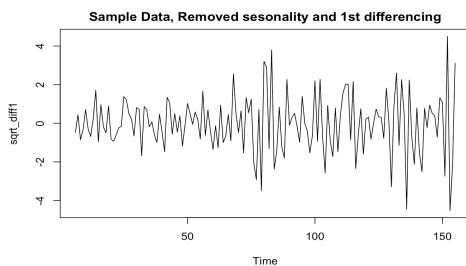
Linear Filter: filter()

```
filter(data, filter=filter_coefficients, sides=2,
       method="convolution", circular=F)
```



Differencing Filter: diff()

```
diff(data, lag=4, differences=1)
```

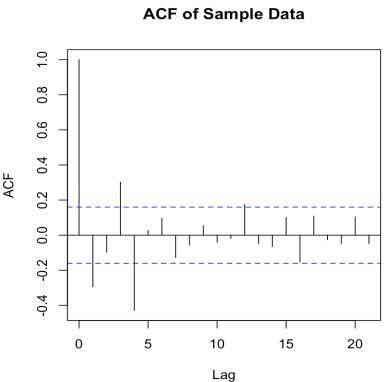


Auto-correlation

Use ACF and PACF to detect model

(Complete) Auto-correlation function: acf()

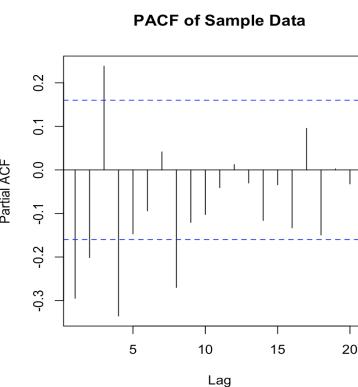
```
acf(data, type='correlation', na.action=na.pass)
```



Partial Auto-correlation function: pacf()

```
pacf(data, na.action=na.pass)
```

OR: acf(data, type='partial', na.action=na.pass)



Forecasting future observations given a fitted ARMA model

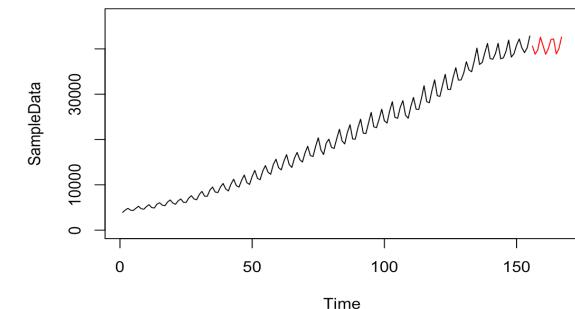
predict(): Predict future observations given a fitted ARMA model

```
predict(arima_model, number_to_predict)
```

Plot Predicted values and Confidence Interval:

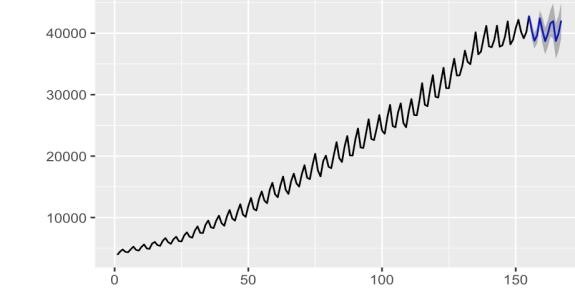
```
fit<-predict(arima_model, number_to_predict)
```

```
ts.plot(data,
        xlim=c(1, length(data)+number_to_predict),
        ylim=c(0, max(fit$pred+1.96*fit$se)))
lines(length(data)+1:length(data)+number_to_predict, fit$pred)
```



OR: autoplot(forecast(arima_model, level=c(95), h=number_to_predict))

Predicted value and Conf Interval of ARIMA





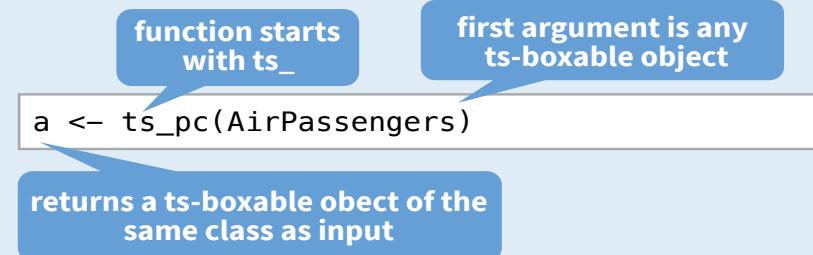
Basics

IDEA

tsbox provides a time series toolkit which:

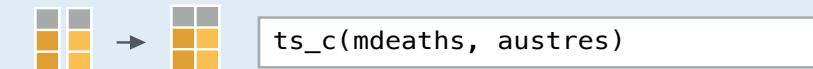
1. works identically with most time series **classes**
2. handles regular and irregular **frequencies**
3. **converts** between classes and frequencies

Most functions in tsbox have the same structure:

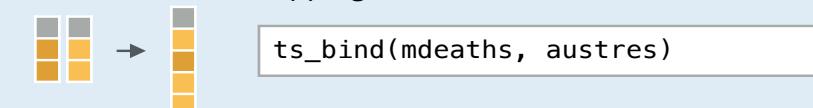


COMBINE TIME SERIES

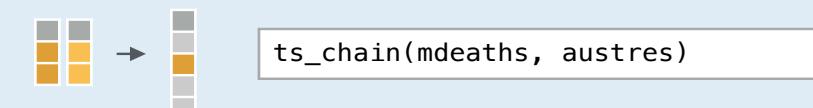
collect time series of **all classes** and **frequencies** as multiple time series



combine time series to a new, single time series (first series wins if overlapping)

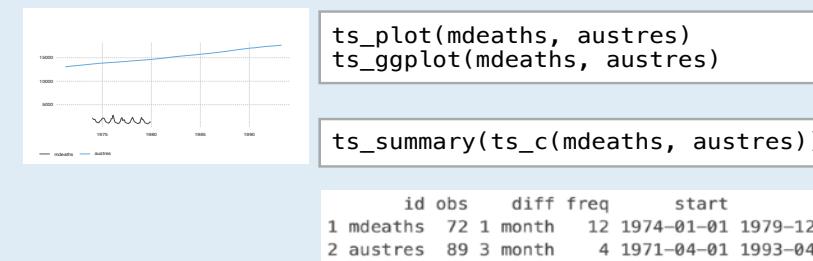


like ts_bind, but extra- and extrapolate, using growth rates



PLOT AND SUMMARIZE

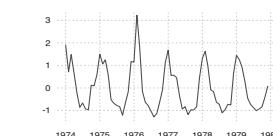
Plot time series of **all classes** and **frequencies**



Helper Functions

Transform time series of **all classes** and **frequencies**

TRANSFORM



ts_trend(): Trend estimation based on loess
`ts_trend(fdeaths)`

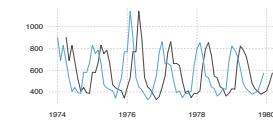
ts_pc(), **ts_pcy()**, **ts_pca()**, **ts_diff()**,
ts_diffy(): (annualized) Percentage change
rates or differences to previous period, year
`ts_pc(fdeaths)`

ts_scale(): normalize mean and variance
`ts_scale(fdeaths)`

ts_index(): Index, based on levels
ts_compound(): Index, based on growth rates
`ts_index(fdeaths, base = 1976)`

ts_seas(): seasonal adjustment using X-13
`ts_seas(fdeaths)`

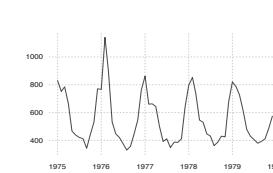
SPAN AND FREQUENCY



ts_lag(): Lag or lead of time series
`ts_lag(fdeaths, 4)`



ts_frequency(): convert to frequency
`ts_frequency(fdeaths, "year")`



ts_span(): filter time series for a time span.
`ts_span(fdeaths, "1976-01-01")`
`ts_span(fdeaths, "-5 year")`

Class Conversion

tsbox is built around a set of converters, which convert time series of the following **supported classes** to each other:

converter function	ts-boxable class
<code>ts_ts()</code>	ts, mts
<code>ts_data.frame()</code> , <code>ts_df()</code>	data.frame
<code>ts_data.table()</code> , <code>ts_dt()</code>	data.table
<code>ts_tbl()</code>	df_tbl, "tibble"
<code>ts_xts()</code>	xts
<code>ts_zoo()</code>	zoo
<code>ts_tibbletime()</code>	tibbletime
<code>ts_timeSeries()</code>	timeSeries
<code>ts_tsibble()</code>	tsibble
<code>ts_tslist()</code>	a list with ts objects

Time Series in data frames

LONG STRUCTURE

Default structure to store multiple time series in long data frames (or data tables, or tibbles)

`ts_df(ts_c(fdeaths, mdeaths))`

id	time	value
fdeaths	1974-01-01	901
fdeaths	1974-02-01	689
fdeaths	1974-03-01	827
...

AUTO-DETECT COLUMN NAMES

tsbox auto-detects a **value**-, a **time**- and zero, one or several **id**-columns. Alternatively, the **time**- and the **value**-column can be explicitly named **time** and **value**.

ts_default(): standardize column names in data frames

RESHAPE

ts_wide(): convert default long structure to wide
ts_long(): convert wide structure to default long

USE WITH PIPE

tsbox plays well with tibbles and with `%>%`, so it can be easily integrated into a dplyr/pipe workflow

```
library(dplyr)
ts_c(fdeaths, mdeaths) %>%
  ts_tbl() %>%
  ts_trend() %>%
  ts_pc()
```

pass return value as first argument to the next function

What is VEGAN?

The **vegan** package provides tools for descriptive community ecology. It has basic functions of **community ordination**, **diversity analysis** and **dissimilarity analysis**. Most of its multivariate tools can be used for other data types as well.

Examples using : **data(dune)**

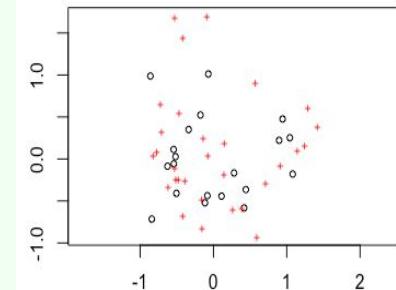
Unconstrained Ordination

metaMDS(data, ...) **Nonmetric Multidimensional Scaling**

All ordination results can be displayed with

plot(data, type = "")

type = "p" results with points of black circles to indicate sites and red pluses to show species

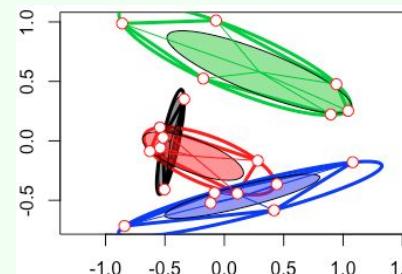


type = "t" results with text

ordihull() adds convex hulls

ordielipse() adds ellipses of standard deviation, standard error or confidence areas

ordispider() draws items to their center



Constrained Ordination

cca(formula, data, ...) **Constrained Correspondence Analysis**
Displays only the variation that can be explained by used constraints

rda(formula, data, scale=FALSE, ...) **Redundancy Analysis**

capscale(formula, data, distance = "", ...) **Distance based Redundancy Analysis**

formula() Model formula must be either community data matrix or dissimilarity matrix

OR

distance = "name of dissimilarity index" if formula is not specified

Analysis of constraints

anova.cca(object, permutations = "", ...) **Permutation Test for CCA & RDA to assess the significance of constraints**

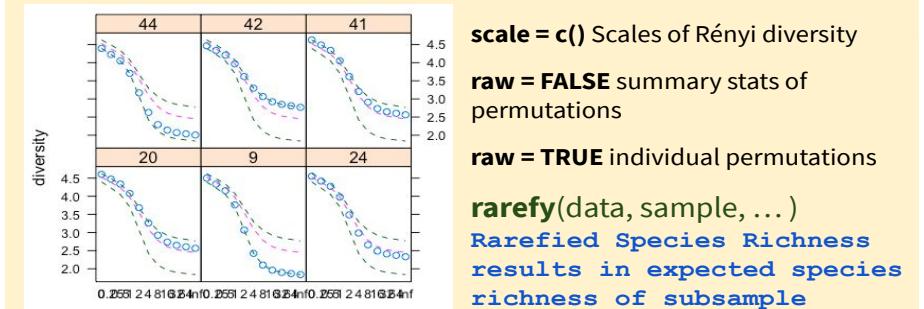
object specifies one or several result objects from cca, rda, or capscale

permutations = control values, or permutation index

Diversity Analysis of Eco Communities

diversity(data, index = "", MARGIN = 1, base = exp(1), ...) **Shannon, Simpson, and Fisher diversity indices and species richness.**

renyi(data, scale = c(), raw = FALSE, ...) **Rényi Diversity index**



scale = c() Scales of Rényi diversity

raw = FALSE summary stats of permutations

raw = TRUE individual permutations

rarefy(data, sample, ...) **Rarefied Species Richness results in expected species richness of subsample**

Taxonomic Diversity

taxondive(data, distance, match.force = FALSE) **Taxonomic diversity indices**

taxa2dist(data, varstep = FALSE, check = TRUE, ...) **Converts class tables to taxonomic distances**

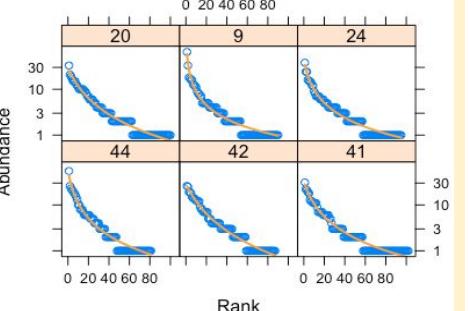
Ranked Abundance Distribution

radfit(data, ...) **Fits the most popular model to data using maximum likelihood estimation**

rad.null(data, family = poisson) **Fits broken stick model to expected abundance of species**

type = "b" Plots both observed points and fitted lines

family = Error distribution; poisson default is used for counts, gaussian may be appropriate for abundance



Beta Diversity

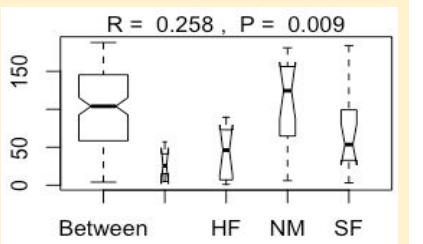
betadiver(data, method = NA, ...) **Estimates beta diversity**

method = "" can specify which beta index to use (24 options)

betadiver(help=TRUE) list all 24 indices available

Analysis of Diversity in Groups

anosim(data, grouping, permutations = "", distance = "", ...) **Analysis of similarities between two or more groups**



Dissimilarity Analysis

vegdist(data, method = "", na.rm = FALSE, ...) **Dissimilarity indices**

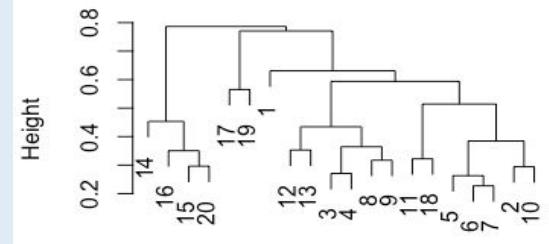
method = "dissimilarity index"

> "manhattan", "euclidean",
"canberra", "clark", "bray",
"kulczynski", "jaccard", "gower",
"altGower", "morisita", "horn",
"mountford", "raup", "binomial",
"chao", "cao" or "mahalanobis".



Other Fun Features

vegemite(data, use, scale, sp.ind = "", site.ind = "", select, ...) **Creates a compact ordered community tree in text format**



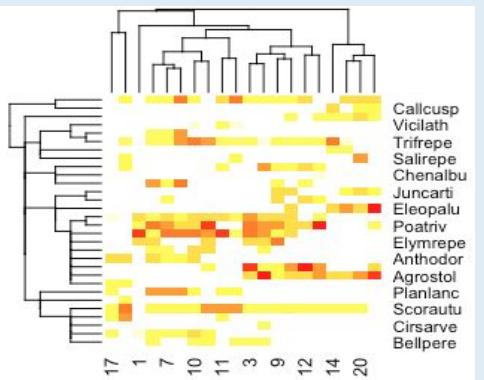
tabasco(data, use, sp.ind = "", site.ind = "", select, Rowv = TRUE, Colv = TRUE, scale, col = heat.colors(12), ...) **Creates a community table using heat map, abundances are coded by color**

use is either a vector or object

sp.ind / site.ind species and site indices

select a subset of plots

Rowv / Colv = reorder rows and columns, if TRUE it is ordered by correspondence analysis



beals(data, species = NA, reference = data, include = TRUE) **Beals Smoothing and Degree of Absence Analysis**

determines probability of a species occurring in a site based on joint occurrences with other species

species = NA will compute for all species, or can specify single

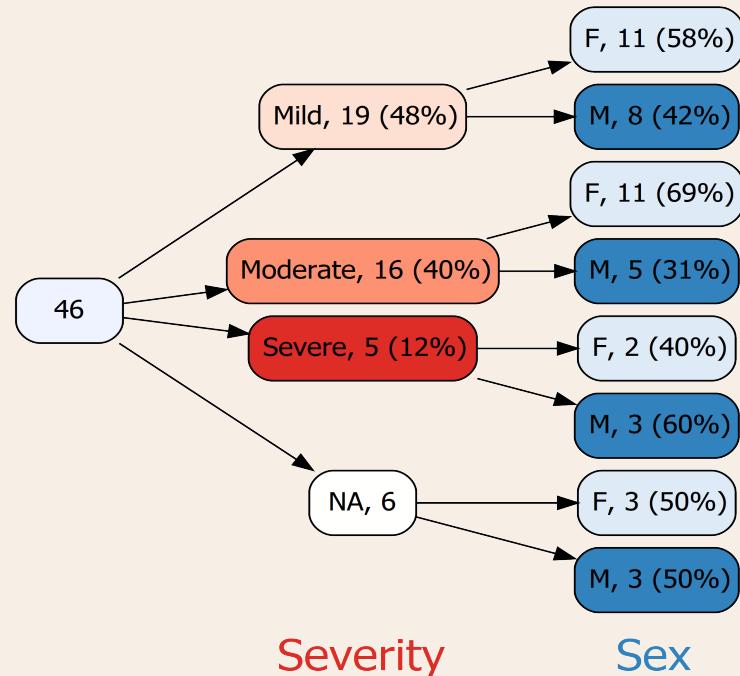
reference = data to be used to compare for joint analysis

include = TRUE to include target species in computations

****VEGAN** uses quantitative data but setting **binary = TRUE** will make data presence/absence**

Examining nested subsets with vtree: cheat sheet

```
vtree(FakeData, "Severity Sex", sameline=T)
```



Pruning

Parameter	Effect
prune	Remove identified nodes and their descendants.
keep	Only retain identified nodes and their descendants.
prunebelow	Remove descendants of identified nodes.
follow	Only retain descendants of identified nodes.

Example: `prune=list(Severity=c("Moderate", "Severe"))`

Labels

Parameter setting	Effect
<code>labelvar=c(variable="Label")</code>	Assign <i>Label</i> to <i>variable</i>
<code>labelnode=list(variable=c(New="Old"))</code>	In <i>variable</i> , replace <i>Old</i> with <i>New</i>
<code>tlabelnode=list(c(Group="A", Sex="F", label="girl"))</code>	Change the label of a specific node
<code>varnamepoints=30</code>	Set font size for variable names
<code>shownodelabels=FALSE</code>	Do not show node labels
<code>showvarnames=FALSE</code>	Do not show variable names
<code>showlegend=TRUE</code>	Show a legend
<code>title="All businesses"</code>	Show a title for the root node

Summaries

Type	Parameter setting
Simple	<code>summary="variable"</code>
Custom	<code>summary="variable format"</code>

Code	Produces
<code>%mean%</code>	mean
<code>%SD%</code>	standard deviation
<code>%sum%</code>	sum
<code>%range%</code>	range
<code>%median%</code>	median
<code>%IQR%</code>	inter-quartile range
<code>%freqpct%</code>	frequency and %
<code>%freq%</code>	just frequency
<code>%npct%</code>	frequency and %
<code>%pct%</code>	%
<code>%list%</code>	list values
<code>%trunc=n%</code>	truncation at <i>n</i> characters

Control code summary restricted to:

<code>%noroot%</code>	all nodes except the root
<code>%leafonly%</code>	leaf nodes
<code>%var=v%</code>	nodes of variable <i>v</i>
<code>%node=n%</code>	nodes named <i>n</i>

Image settings

Parameter setting	Effect
<code>imagewidth="3in"</code>	3 inches wide
<code>imageheight="4in"</code>	4 inches tall
<code>pxwidth=800</code>	800 pixels wide
<code>pxheight=2000</code>	200 pixels high

Frequencies and percentages

Parameter setting	Effect
<code>vp=FALSE</code>	Full denominator
<code>showpct=FALSE</code>	Do not show %
<code>showcount=FALSE</code>	Do not show counts

Variable specification

Suffix	Effect
#	Variable names ending in numeric digits
*	Variable names ending in any character
@	REDCap checklist variable names

Prefix	Effect
<code>is.na:</code>	Missing value?
<code>r:</code>	REDCap checklist variable
<code>i:</code>	Intersection of group of variables
<code>any:</code>	Are any of a group of variables affirmative?
<code>all:</code>	Are all of a group of variables affirmative?

Text

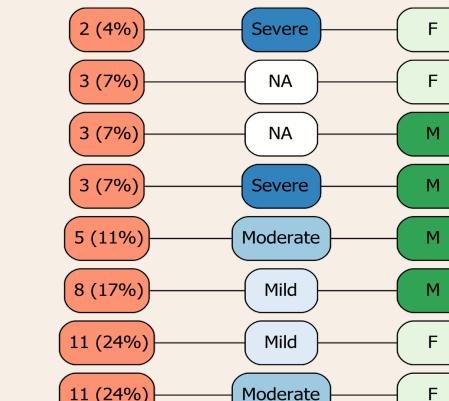
Parameter setting	Text
<code>text=list(Category=c(triple="*"))</code>	Add * to all nodes of this type
<code>ttext=list(c(Group="A", Category="triple", text="*"))</code>	Add * to a specific node

Formatting

`\n` line break `*italics*` `**bold**` `%red ...%`

Pattern trees and tables

`vtree(FakeData, "Severity Sex", pattern=T, varnamebold=T)` `vtree(FakeData, "Severity Sex", ptable=T)`



n	pct	Severity	Sex
2	4	Severe	F
3	7	<NA>	F
3	7	<NA>	M
3	7	Severe	M
5	11	Moderate	M
8	17	Mild	M
11	24	Mild	F
11	24	Moderate	F

Splitting text across lines

Parameter setting	Effect
<code>splitwidth=50</code>	Split text in nodes after 50 characters
<code>vsplitwidth=5</code>	Split text in variable names after 5 characters

xplain Cheat Sheet

Important Links

- xplain package on CRAN <https://cran.r-project.org/web/packages/xplain/index.html>
- xplain web tutorial <http://www.zuckarelli.de/xplain/index.html>
- xplain cheat sheet http://www.zuckarelli.de/xplain/xplain_cheatsheet.pdf
- xplain on GitHub <https://www.github.com/jsugarelli/xplain>

Purpose & Application

- xplain allows to **write interpretation/explanation texts** for statistical functions in the form of XML files.
- The user of the functions can read these explanations **while working on his/her specific problems**.
- xplain explanations **can react to the user's results** and provide meaningful insights related to the user's problem.
- For this, the xplain **XML files can contain R code** and can **work with the return object** of the user's function call.

```
> xplain("lm(education ~ young + income + urban)")
> Your R^2 is 0.11 which is quite low. There is a serious
risk your model is misspecified. You should reconsider the
selection of variables included in your model.
```

xplain XML files

1 Any valid xplain XML must be enclosed in an `<xplain>` block. Multiple `<xplain>` blocks per XML file are possible.

2 A `<package>` block combines all functions from the same package.

3 Within a `<function>` block, explanations/interpretations for the function as such or for specific elements of the return object can be provided.

4 Packages explanations/ interpretations related to one element of the function's return object.

```
<xml>
  1 <xplain>
    2 <package name = "stats">
      3 <function name = "lm">
        4 <title>This is about lm</title>
        5 <text>...</text>
        6 <result name = "coefficients">
          4 <title>...</title>
          5 <text>...</text>
        </result>
      </function>
    </package>
  </xplain>
</xml>
```

Not case-sensitive

5 Structures explanations with headers.

`<text>`

6 The actual explanations/interpretations. Can include R code with references to the function's return object.

Main attributes: Overview

name	Name of the element (package, function, result).
lang	Language (ISO code) of the explanation (e.g. "EN").
level	Complexity level; integer number; cumulative, i.e. <code>level=1</code> explanations will also be presented when <code>level=2</code> or <code>level=3</code> are called.

Attributes: Inheritance and necessity

- Elements **inherit attributes from higher-level elements**; e.g., if only one language, definition on `<xplain>` level suffices. Lower-level attributes overrule higher-level.
- **name** attribute required for `<package>`, `<function>` and `<result>` elements.
- All levels shown, if no `level` is given to `xplain()`.

Including R code

R code can be easily integrated into `<text></text>` elements:

```
<text> !%< R code %!> </text>
      ↑           ↑
      R code delimiter tags
```

Access the explained function's (`<function name="...">`) return object:

- Access the full return object with `@`. Example: `summary(@)`.
- Access the current `<result name="...">` item of the return object with `##`. Example: `mean(##)`.

Using placeholders

```
<define name= "placeholder" > !%< R code %!> </define>
      ↓
</text> Text... !** "placeholder" **! Text... </text>
      ↑           ↑
      Placeholder name delimiter tags
```

Example: `<define name="s">!%< summary(@) %!> </define>`
`<text>And here is the summary !**s**! for your model</text>`

Iterating through (items of) the return object

- To apply a `<text>` element to a whole matrix, data frame, vector or list, use the **foreach** attribute.
- Value of foreach defines what is iterated over and (for 2D structures) in which sequence; `items` is for lists.
- `$` is a placeholder for the index of the current element.
- **Example** (shows all 1st column elements of the coefficient matrix):
`<text foreach="rows">!%< @$coefficients[,$1] %!> </text>`

```
foreach =
"rows"
"columns"
"rows, columns"
"columns, rows"
"items"
```

Calling xplain()

1	<code>call</code>	Call of the explained function as string
Direct call of <code>xplain()</code>	<code>xml</code>	Path of the XML file providing the explanations
	<code>lang</code>	Language of the explanations to be shown (default means English)
	<code>level</code>	Complexity level of the explanations (cumulative! Default means "all")

- 2**
- Wrapper function with `xplain.getcall()`

Example: lm

```
lm.xplain <- function(formula, data, subset, weights, na.action,
method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
singular.ok = TRUE, contrasts = NULL, offset, ...) {
  call<-xplain.getcall("lm")
  xplain(call, xml="http://www.zuckarelli.de/example_lm.xml")
}
```