

Four Column Layout :: CHEAT SHEET

Basics

Thank you for making a new cheatsheet for R!
These cheatsheets have an important job:

**Cheatsheets make it easy for R users
to look up useful information.**

Remember that the best cheatsheets are **visual**—not written—documents. Whenever possible use visual elements to make it easier for readers to find the information they need.

1. Use a **layout** that flows and makes it easy to zero in on specific topics.



2. Use **visualizations** to explain concepts quickly and concisely.

summary function →

3. Use visual elements to make the sheet **scannable**.

i + geom_area()
x, y, alpha, color, fill, linetype, size

i + geom_line()
x, y, alpha, color, group, linetype, size

4. Use visual **emphasis** (like color, size, and font weight) to make important information easy to find.

dplyr::lag() - Offset elements by 1
dplyr::lead() - Offset elements by -1

Layout Suggestions

Use headers, colors, and/or backgrounds to **separate or group together sections**.

Section 1 Section 2 Section 3

Create a visual hierarchy. Help users navigate the page with titles, subtitles, and subsubtitles

Title

SUBTITLE

SUBSUBTITLE

Fit sections to content. Try several different layouts.

Use numbers or arrows to link sections if the order/**flow** is confusing.

Quickly identify content with a **package hexsticker** (if available)



Copyright

Each cheatsheet should be licensed under the creative commons license.

To license the sheet as creative commons, put CC'd by <your name> in the small print at the bottom of each page and link it to <http://creativecommons.org/licenses/by/4.0/>

YOUR LOGO
(optional)

Useful Elements

CODE

Where possible, use **code that works** when run.

```
ggplot(mpg, aes(hwy, cty)) +  
  geom_point(aes(size = fl)) +  
  geom_smooth(method = "lm")
```

Word balloons

can help explain

code

ICONS

These are just **font awesome** characters



MOCK TABLES



MOCK GRAPHS



TABLES

sub-option	description
citation_package	The LaTeX package to process
code_folding	Let readers to toggle the display of R
colortheme	Beamer color theme to use

Logistics

FONTS

This template uses several fonts:
Helvetica Neue, Menlo, Source Sans Pro, which you can acquire for free here, www.fontsquirrel.com/fonts/source-sans-pro, and **Font Awesome**, which you can acquire here, fortawesome.github.io/Font-Awesome/get-started/

To use a **font awesome** icon, copy and paste one from here fortawesome.github.io/Font-Awesome/cheatsheet/. Then set the text font to font awesome.

KEYNOTE

I make my cheatsheets in **Apple Keynote**, and not latex or R Markdown, because presentation software makes it much easier to tweak the visual appearance of a document

KEYNOTE TIPS

- **Select multiple elements** by holding down shift and then selecting each. Click on a selected element before letting go of shift to unselect it.
- To **group elements together**. Select them all, then click Arrange > Group
- To **evenly space multiple objects**, select them all then Right Click > Align objects or Right Click > Distribute objects
- Click on a table, then visit Format >Table > Row and Column Size to make **even width rows/columns**.

Three Column Layout: : CHEAT SHEET



Basics

Thank you for making a new cheatsheet for R! These cheatsheets have an important job:

Cheatsheets make it easy for R users to look up useful information.

Remember that the best cheatsheets are **visual**—not written—documents. Whenever possible use visual elements to make it easier for readers to find the information they need.

1. Use a **layout** that flows and makes it easy to zero in on specific topics.



2. Use **visualizations** to explain concepts quickly and concisely.



3. Use visual elements to make the sheet **scannable**.



4. Use visual **emphasis** (like color, size, and font weight) to make important information easy to find.

`dplyr::lag()` - Offset elements by 1
`dplyr::lead()` - Offset elements by -1

COPYRIGHT

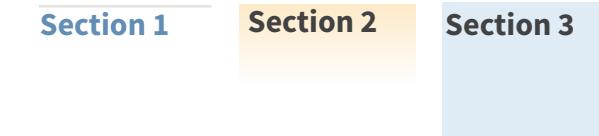
Each cheatsheet should be licensed under the creative commons license.

To license the sheet as creative commons, put CC'd by <your name> in the small print at the bottom of each page and link it to <http://creativecommons.org/licenses/by/4.0/>

YOUR LOGO
(optional)

Layout Suggestions

Use headers, colors, and/or backgrounds to **separate or group together sections**.



Logistics

FONTS

This template uses several fonts: **Helvetica Neue**, **Menlo**, **Source Sans pro**, which you can acquire for free here, www.fontsquirrel.com/fonts/source-sans-pro, and **Font Awesome**, which you can acquire here, [fortawesome.github.io/Font-Awesome/get-started/](https://fontawesome.github.io/Font-Awesome/get-started/)

To use a **font awesome** icon, copy and paste one from here fortawesome.github.io/Font-Awesome/cheatsheet/. Then set the text font to font awesome.

KEYNOTE

I make my cheatsheets in **Apple Keynote**, and not latex or R Markdown, because presentation software makes it much easier to tweak the visual appearance of a document

KEYNOTE TIPS

- **Select multiple elements** by holding down shift and then selecting each. Click on a selected element before letting go of shift to unselect it.
- **To group elements together.** Select them all , then click Arrange > Group
- **To evenly space multiple objects**, select them all then Right Click > Align objects or Right Click > Distribute objects
- Click on a table, then visit Format >Table > Row and Column Size to make **even width rows/columns**.

Manipulate Variables

Create a visual hierarchy: Help users navigate the page with titles, subtitles, and subsubtitles

Title

SUBTITLE

SUBSUBTITLE

Quickly identify content with a **package hexsticker** (if available)

Fit sections to content. Try several different layouts.

Use numbers or arrows to link sections if the order/**flow** is confusing.

Useful Elements

CODE

Where possible, use **code that works** when run.

```
ggplot(mpg, aes(hwy, cty)) +  
  geom_point(aes(color = cyl)) +  
  geom_smooth(method = "lm")
```

Word balloons

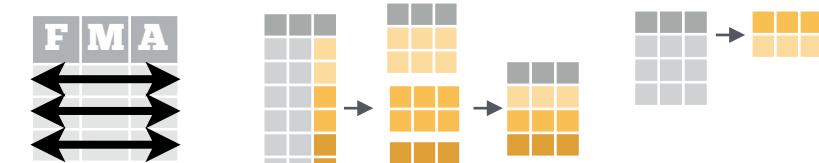
can help explain code

ICONS



These are just font awesome characters

MOCK TABLES



MOCK GRAPHS



TABLES

sub-option	description
citation_package	The LaTeX package to process
code_folding	Let readers to toggle the display of R
colortheme	Beamer color theme to use

Machine Learning Modelling in R :: CHEAT SHEET

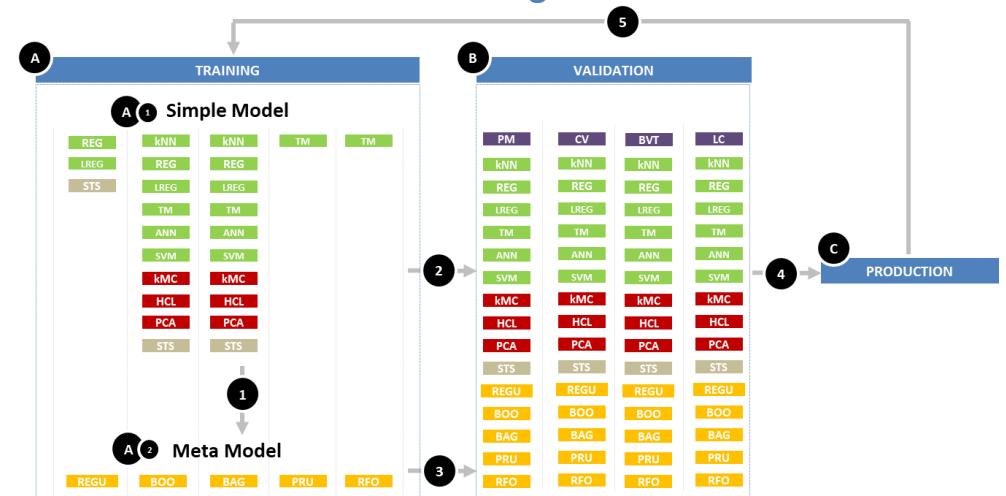
Supervised & Unsupervised Learning

ALGORITHM	DESCRIPTION	R PACKAGE::FUNCTION	SAMPLE CODE
NBC 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 or absence of a particular feature in a class is unrelated to the presence of any other feature	e1071::naiveBayes	naiveBayes(class ~ ., data = x)
kNN 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)
LRG 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)
LRC 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 Tree-Based Models	The idea is to consecutively divide (branch) the training data into smaller and smaller 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 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=(5,3),linear.output=T)
SVM 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 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 stats::princomp FactoMineR::PCA ade4::dudi.pca amap::acp	stats::prcomp(formula, data = NULL, subset, na.action, ...) stats::princomp(formula, data = NULL, subset, na.action, ...) FactoMineR::PCA(decatlon, quanti.sup = 11:12, quali.sup = 13) ade4::dudi.pca(deugStab, center = deugCent, scale = FALSE, scan = FALSE) amap::acp(lubisch)
HAC 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(k, centers, iter.max = 10, nstart = 1, algorithm = c("Hartigan-Wong", "Lloyd", "Forgy", "MacQueen"), trace = FALSE)
HCL 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)

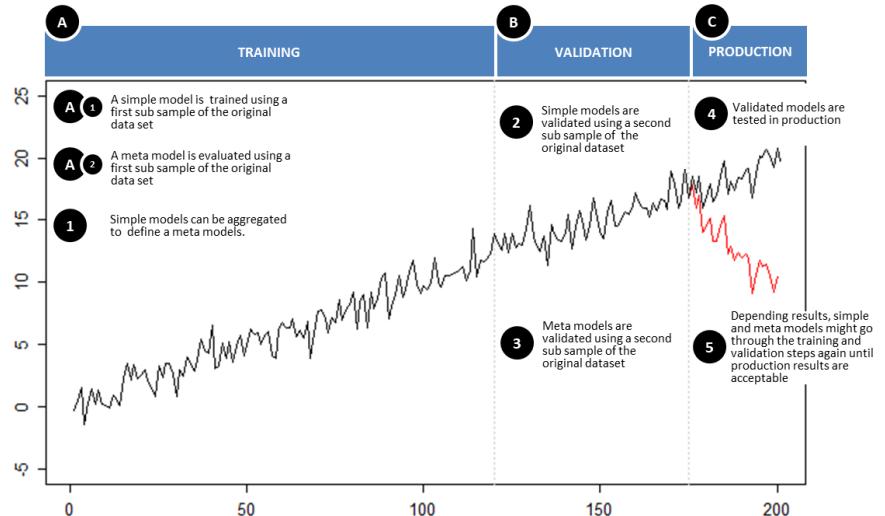
Meta-Algorithm, Time Series & Model Validation

ALGORITHM	DESCRIPTION	R PACKAGE::FUNCTION	SAMPLE CODE
REGU Regularisation L1 (Lasso) L2 (Ridge)	Regularisation 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) L2 : glmnet(myMatrixA, myMatrixB, family = "gaussian", alpha = 0)
BOO 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.	gbm::gbm	gbmboost(Y ~ ., data = curr1[trnidxs,])
BAG 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	randomForest::randomForest	foreach : d <- data.frame(x=1:10, y=rnorm(10)) s <- foreach(d=i %in% d, by=row, .combine=rbind, .id=i, .d = i) ipred : bagging(formula, data, subset, na.action=na.rpart, \dots)
PRU 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 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 Time Series	Random sampling of observations for training and testing a model can be an issue when faced with a times dimension. Random sampling may either destroy serial correlation properties in the data which we would like to exploit	stats::xts forecast::spectral spectral::TTR	Auto-correlation: acf(x, lag.max = NULL, type = c("correlation", "covariance", "partial")) Spectral Analysis: spec.pgram(..., spans = NULL) Seasonal Decomposition of Time Series : stl(x, s.window = 7, t.window = 50, t.jump = 1)
PM Performance metrics	Depends on the problem: • Regression: squared errors, outliers, error rate... • Classification: Accuracy, precision, recall, F-score...	ROCR::ROC	Regression:stats::outlierTest, stats::qqPlot ... Classification:ROCR::ROC
JVT 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)	caret::confusionMatrix	Tailored to the analysis
CV Cross validation	Cross validation compares the test performances of different model realisations with different sets or values of parameters	caret::createDataPartition caret::createFolds	Tailored to the analysis
LC 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	createDataPartition(classes, p = 0.8, list = FALSE) learning_curve_dat(dat, outcome = NULL, proportion = (1:10)/10, test_prop = 0, verbose = TRUE, ...)

Standard Modelling Workflow



Time Series View



SamplingStrata: : CHEAT SHEET

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
```



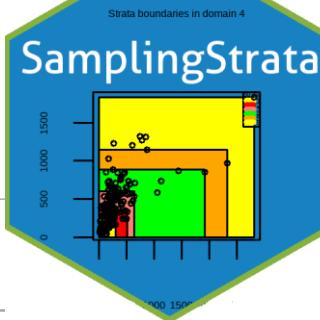
Sample selection

```
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

To install last available release:

```
library(devtools)
install_github("barcaroli/SamplingStrata")
```



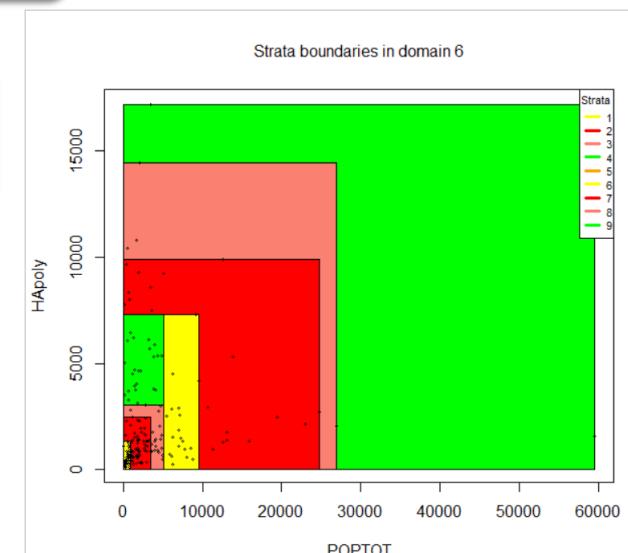
Evaluation

```
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

Lower_X1	Upper_X1	Lower_X2	Upper_X2	
1	27	780	32	986
2	65	1422	48	1018
3	95	1562	198	2288
4	78	1963	159	11907
5	1992	2711	107	8925
6	2759	3567	185	11378

```
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

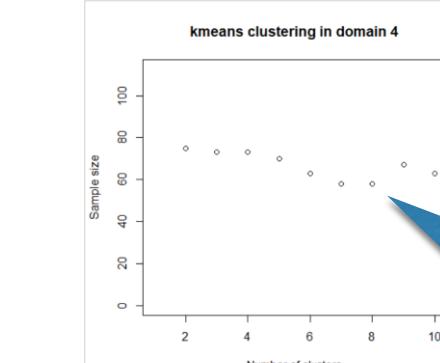
B. Method "continuous"

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

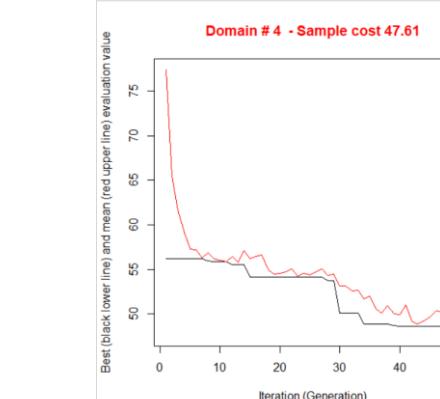
```
kmean <- KmeansSolution2(frame=frame,
                           errors=cv,
                           maxclusters = 10)
nstrat <- tapply(kmean$suggestions,
                  kmean$domainvalue,
                  FUN=function(x)
                    length(unique(x)))
sugg <- prepareSuggestion(
  kmean = kmean,
  frame = frame,
  nstrat = nstrat)
```



Visualization of strata by couples of X's
Suggested number of strata (8) for domain 4

Optimization

```
solution <- optimStrata (
  method = "continuous",
  framesamp = frame,
  errors = cv,
  nStrata = nstrat,
  iter = 50,
  pops = 10,
  suggestions = sugg)
```



C. Method "spatial"

In cases where units in the sampling frame are 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.

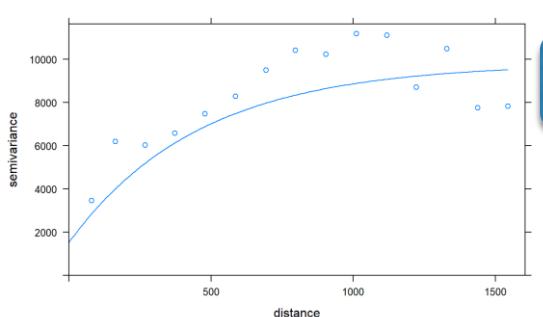
Spatial analysis

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')
```



```
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)
```



```
prediction
lead.kr <- krige(lead~dist+soil,
  meuse, meuse.grid,
  model=fit.vgm.lead$var_model)
lead.pred <- ifelse(lead.kr[1]$var1.pred<0,
  0,lead.kr[1]$var1.pred)
lead.var <- ifelse(lead.kr[2]$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))
```



Use of models

Usually, values of target variables are not available in sampling frames, but only of co-variates. In order 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
```

Sampling frame

```
frame <- buildFrameDF(
  df=swissmunicipalities,
  id="id",
  X=c("POPTOT", "HApoly"),
  Y=c("POPTOT", "HApoly"),
  domainvalue = "REG")
```

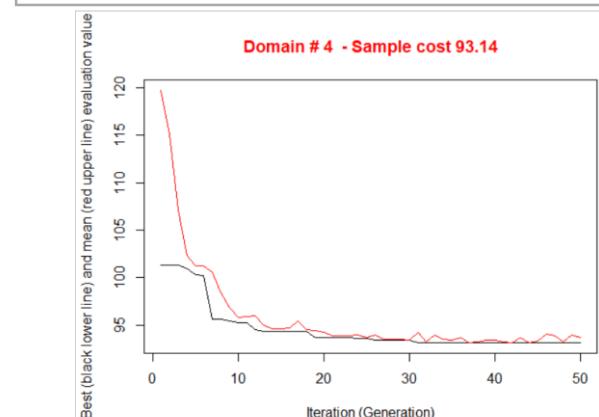
```
frame$airind <-
  swissmunicipalities$Airind
frame$surfacesbois <-
  swissmunicipalities$Surfacesbois
```

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.

Base R Cheat Sheet

Getting Help

Accessing the help files

?mean

Get help of a particular function.

help.search('weighted mean')

Search the help files for a word or phrase.

help(package = 'dplyr')

Find help for a package.

More about an object

str(iris)

Get a summary of an object's structure.

class(iris)

Find the class an object belongs to.

Using Packages

install.packages('dplyr')

Download and install a package from CRAN.

library(dplyr)

Load the package into the session, making all its functions available to use.

dplyr::select

Use a particular function from a package.

data(iris)

Load a built-in dataset into the environment.

Working Directory

getwd()

Find the current working directory (where inputs are found and outputs are sent).

setwd('C://file/path')

Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

Vectors

Creating Vectors

c(2, 4, 6)	2 4 6	Join elements into a vector
2:6	2 3 4 5 6	An integer sequence
seq(2, 3, by=0.5)	2.0 2.5 3.0	A complex sequence
rep(1:2, times=3)	1 2 1 2 1 2	Repeat a vector
rep(1:2, each=3)	1 1 1 2 2 2	Repeat elements of a vector

Vector Functions

sort(x)

Return x sorted.

rev(x)

Return x reversed.

table(x)

See counts of values.

unique(x)

See unique values.

Selecting Vector Elements

By Position

x[4]

The fourth element.

x[-4]

All but the fourth.

x[2:4]

Elements two to four.

x[!(2:4)]

All elements except two to four.

x[c(1, 5)]

Elements one and five.

By Value

x[x == 10]

Elements which are equal to 10.

x[x < 0]

All elements less than zero.

x[x %in% c(1, 2, 5)]

Elements in the set 1, 2, 5.

Named Vectors

x['apple']

Element with name 'apple'.

Programming

For Loop

```
for (variable in sequence){  
  Do something  
}
```

Example

```
for (i in 1:4){  
  j <- i + 10  
  print(j)  
}
```

While Loop

```
while (condition){  
  Do something  
}
```

Example

```
while (i < 5){  
  print(i)  
  i <- i + 1  
}
```

Functions

```
function_name <- function(var){  
  Do something  
  return(new_variable)  
}
```

Example

```
square <- function(x){  
  squared <- x*x  
  return(squared)  
}
```

Reading and Writing Data

Also see the **readr** package.

Input	Output	Description
df <- read.table('file.txt')	write.table(df, 'file.txt')	Read and write a delimited text file.
df <- read.csv('file.csv')	write.csv(df, 'file.csv')	Read and write a comma separated value file. This is a special case of read.table/write.table.
load('file.RData')	save(df, file = 'file.Rdata')	Read and write an R data file, a file type special for R.

Conditions	a == b	Are equal	a > b	Greater than	a >= b	Greater than or equal to	is.na(a)	Is missing
	a != b	Not equal	a < b	Less than	a <= b	Less than or equal to	is.null(a)	Is null

Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

as.logical	TRUE, FALSE, TRUE	Boolean values (TRUE or FALSE).
as.numeric	1, 0, 1	Integers or floating point numbers.
as.character	'1', '0', '1'	Character strings. Generally preferred to factors.
as.factor	'1', '0', '1', levels: '1', '0'	Character strings with preset levels. Needed for some statistical models.

Maths Functions

log(x)	Natural log.	sum(x)	Sum.
exp(x)	Exponential.	mean(x)	Mean.
max(x)	Largest element.	median(x)	Median.
min(x)	Smallest element.	quantile(x)	Percentage quantiles.
round(x, n)	Round to n decimal places.	rank(x)	Rank of elements.
signif(x, n)	Round to n significant figures.	var(x)	The variance.
cor(x, y)	Correlation.	sd(x)	The standard deviation.

Variable Assignment

```
> a <- 'apple'  
> a  
[1] 'apple'
```

The Environment

ls()	List all variables in the environment.
rm(x)	Remove x from the environment.
rm(list = ls())	Remove all variables from the environment.

You can use the environment panel in RStudio to browse variables in your environment.

Matrices

`m <- matrix(x, nrow = 3, ncol = 3)`
Create a matrix from x.

	<code>m[2,]</code> - Select a row	<code>t(m)</code> Transpose
	<code>m[, 1]</code> - Select a column	<code>m %*% n</code> Matrix Multiplication
	<code>m[2, 3]</code> - Select an element	<code>solve(m, n)</code> Find x in: $m \cdot x = n$

Lists

`l <- list(x = 1:5, y = c('a', 'b'))`
A list is a collection of elements which can be of different types.

<code>l[[2]]</code>	<code>l[1]</code>	<code>l\$x</code>	<code>l['y']</code>
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

Also see the `dplyr` package.

Data Frames

`df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))`
A special case of a list where all elements are the same length.

x	y
1	a
2	b
3	c

Matrix subsetting

<code>df[, 2]</code>		<code>nrow(df)</code> Number of rows.	<code>cbind</code> - Bind columns.
<code>df[2,]</code>		<code>ncol(df)</code> Number of columns.	<code>rbind</code> - Bind rows.
<code>df[2, 2]</code>		<code>dim(df)</code> Number of columns and rows.	

Strings

<code>paste(x, y, sep = ' ')</code>	Join multiple vectors together.
<code>paste(x, collapse = ' ')</code>	Join elements of a vector together.
<code>grep(pattern, x)</code>	Find regular expression matches in x.
<code>gsub(pattern, replace, x)</code>	Replace matches in x with a string.
<code>toupper(x)</code>	Convert to uppercase.
<code>tolower(x)</code>	Convert to lowercase.
<code>nchar(x)</code>	Number of characters in a string.

Factors

<code>factor(x)</code>	Turn a vector into a factor. Can set the levels of the factor and the order.
<code>cut(x, breaks = 4)</code>	Turn a numeric vector into a factor by 'cutting' into sections.

Statistics

<code>lm(y ~ x, data=df)</code>	Linear model.	<code>t.test(x, y)</code>	Test for a difference between proportions.
<code>glm(y ~ x, data=df)</code>	Generalised linear model.	<code>pairwise.t.test</code>	Perform a t-test for paired data.
<code>summary</code>	Get more detailed information out a model.	<code>aov</code>	Analysis of variance.

Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	<code>rnorm</code>	<code>dnorm</code>	<code>pnorm</code>	<code>qnorm</code>
Poisson	<code>rpois</code>	<code>dpois</code>	<code>ppois</code>	<code>qpois</code>
Binomial	<code>rbinom</code>	<code>dbinom</code>	<code>pbinom</code>	<code>qbinom</code>
Uniform	<code>runif</code>	<code>dunif</code>	<code>unif</code>	<code>qunif</code>

Plotting

<code>plot(x)</code>	Values of x in order.	<code>plot(x, y)</code>	Values of x against y.	<code>hist(x)</code>	Histogram of x.
----------------------	-----------------------	-------------------------	------------------------	----------------------	-----------------

Dates

See the `lubridate` package.

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.

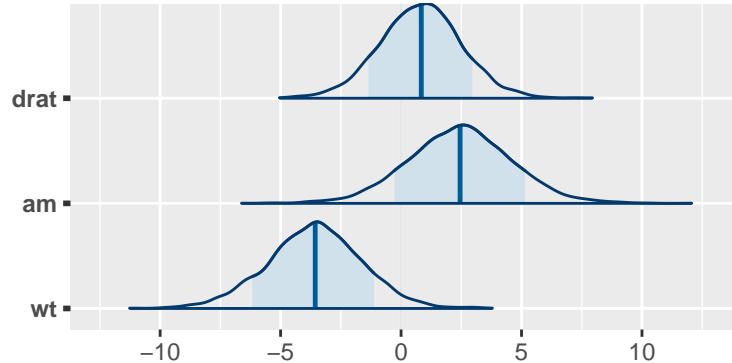
```
model <- stan_glm(mpg ~ ., data=mtcars, chains=4)
posterior <- as.matrix(model)
```

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
```

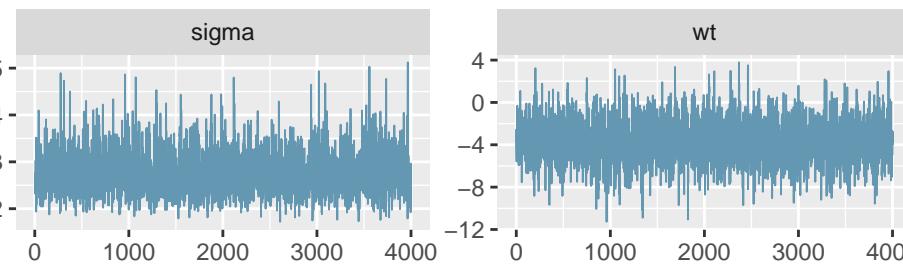
Posterior distributions

medians and 80% intervals



Diagnosing convergence with traceplots is simple.

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



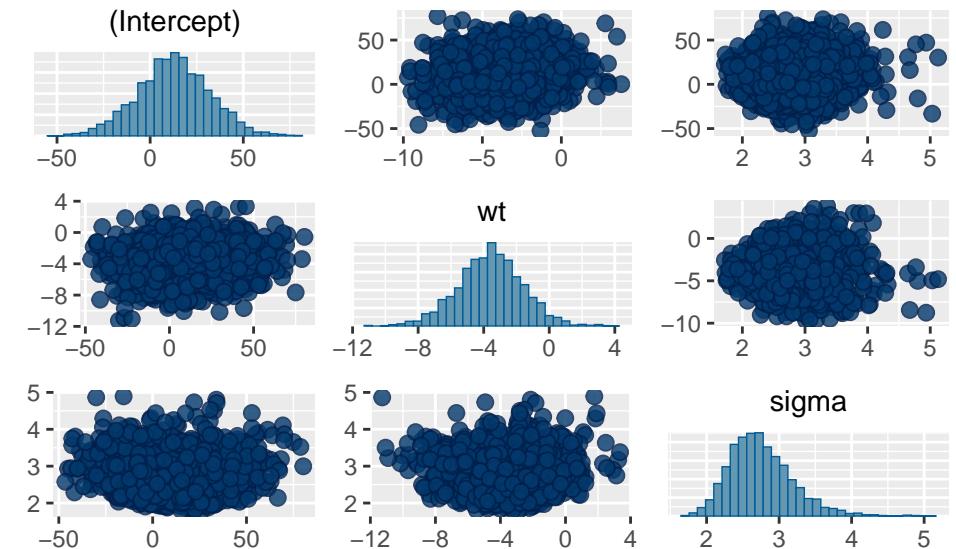
Using `as.array`, you can extract each of the four chain's posterior draws, different from above. This allows you to see each chain's traceplot for selected parameters.

```
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"))
```

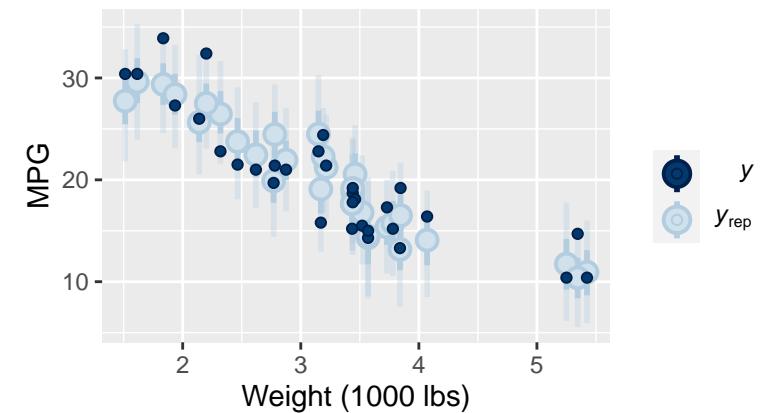


Posterior Predictive Checks

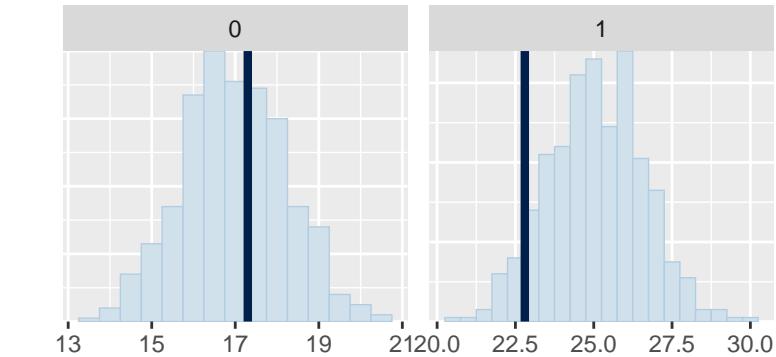
Check how well the model covers your data with draws from the posterior predictive density.

```
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)
```



$T = \text{median}$
 $T(y_{\text{rep}})$
 $T(y)$

Diagnostics

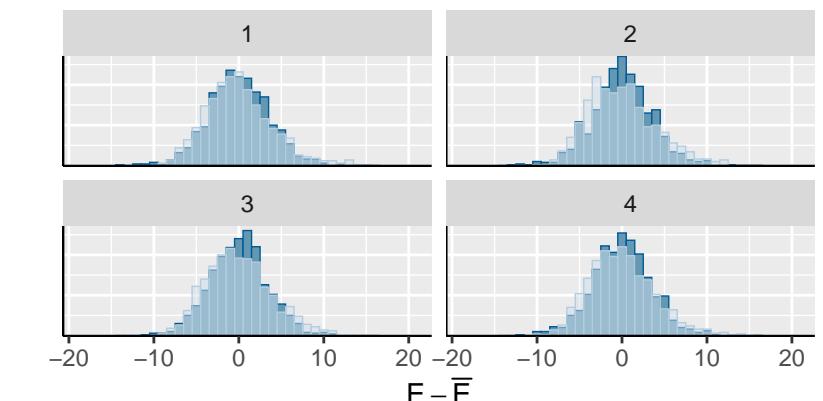
Bayesplot makes it easy to check diagnostics specific to the NUTS sampling method that `rstanarm` uses by default.

```
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



π_E
 $\pi_{\Delta E}$

BCEA :: CHEAT SHEET



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



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`

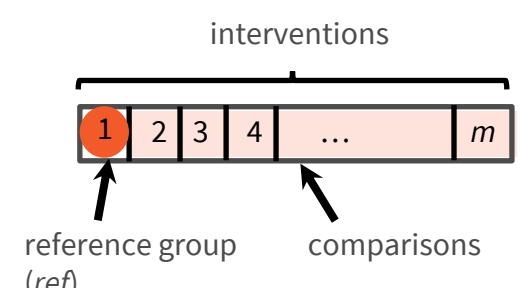
3. *In plotting call*: At the point of making a plot.

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

SELECTING ANALYSIS INTERVENTIONS

Default

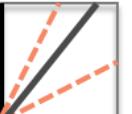
The first columns in (*eff*, *cost*) are the default reference intervention. All other interventions are the comparison interventions unless defined otherwise. E.g. for *m* interventions



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

caret Package

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.

Preprocessing

Transformations, filters, and other operations can be applied to 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).
- `"L0O"` 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`).

Performance Metrics

To choose how to summarize a model, the `trainControl` 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.

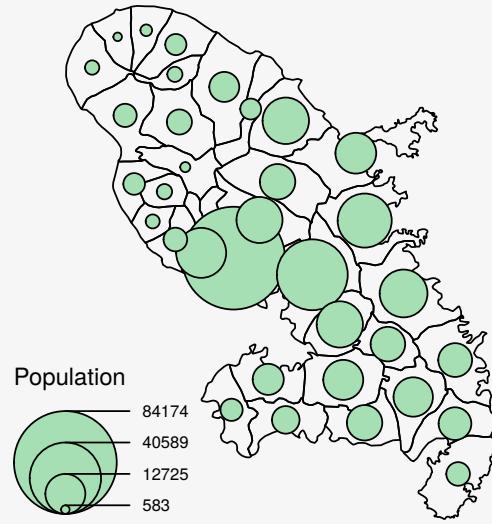
```
trainControl(sampling = "down")
```

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",
typefc = "exponential", span = 500,
beta = 2)



Discontinuities
discLayer(x = mtq.borders, df = mtq_df,
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

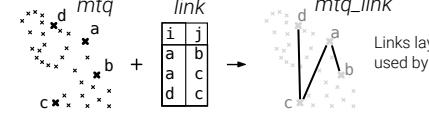
```
mtq_grid <- getGridLayer(x = mtq, cellsize = 3.6e+07,
  type = "hexagonal", var = "myvar")
```



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

Points to Links

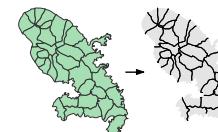
```
mtq_link <- getLinkLayer(x = mtq, df = link)
```



Links layers can be
used by *LinkLayer().

Polygons to Borders

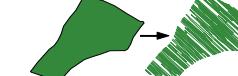
```
mtq_border <- getBorders(x = mtq)
```



Borders layers can be used by
discLayer() function

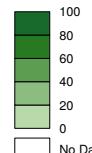
Polygons to Pencil Lines

```
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", 6),  
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()

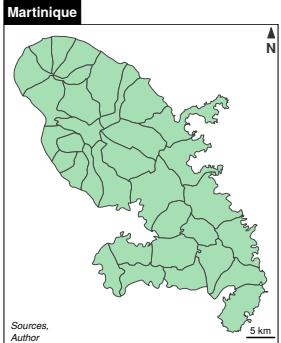


```
legendCirclesSymbols(var = c(10,100),  
title.txt = "legendCirclesSymbols()",  
col = "#a7dfb4ff", inches = 0.3)
```

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

Map Layout

North Arrow:
north(pos = "topright")



Scale Bar:
barscale(size = 5)

Full Layout:

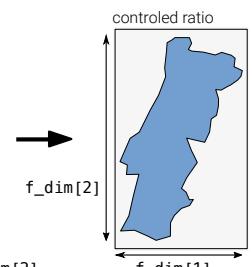
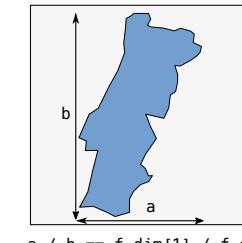
```
layoutLayer(
  title = "Martinique",
  tabtitle = TRUE,
  frame = TRUE,
  author = "Author",
  sources = "Sources",
  north = TRUE,
  scale = 5)
```

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()
```

default



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, ffirst, 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

```
flm(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  
qG(), 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
```

```
qM() – to matrix, m[r/c]t() – 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(), pwnobs() – 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...
```



Data import with the tidyverse :: CHEAT SHEET

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;5;NA
A	B	C
1	5	2
4	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"))`

A	B	C
NA	2	3
4	5	NA

Read multiple files into a single table

`read_csv(c("f1.csv", "f2.csv", "f3.csv"),
id = "origin_file")`

Save Data with readr

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

A	B	C
1	2	3
4	5	NA

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()** - "?"

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"
)
```

Import Spreadsheets with readxl

READ EXCEL FILES

	A	B	C	D	E
1	x1	x2	x3	x4	x5
2	x		z	8	
3	y	7		9	10

```
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

```
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**



READXL COLUMN SPECIFICATION

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

Use the **col_types** argument of **read_excel()** 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

```
read_excel(  
  path,  
  col_types = c("text", "guess", "guess", "numeric"))
```

COLUMN TYPES

logical	numeric	text	date	list
TRUE	2	hello	1947-01-08	hello
FALSE	3.45	world	1956-10-21	1

- skip
- guess
- logical
- numeric
- date
- list
- text

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

CELL SPECIFICATION FOR READXL AND GOOGLESHEETS4

A	B	C	D	E
1	1	2	3	4
2	x		y	z
3	6	7		9

Use the **range** argument of **readxl::read_excel()** or **googlesheets4::read_sheet()** to read a subset of cells from a sheet.

```
read_excel(path, range = "Sheet1!B1:D2")  
read_sheet(ss, range = "B1:D2")
```

Also use the range argument with cell specification functions **cell_limits()**, **cell_rows()**, **cell_cols()**, and **anchored()**.



with googlesheets4

READ SHEETS

	A	B	C	D	E
1	x1	x2	x3	x4	x5
2	x		z	8	
3	y	7		9	10

```
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

1	x	4
2	y	5
3	z	6

write_sheet(data, ss = NULL, sheet = NULL)
Write a data frame into a new or existing Sheet.

gs4_create(name, ..., sheets = NULL) Create a new Sheet with a vector of names, a data frame, or a (named) list of data frames.

sheet_append(ss, data, sheet = 1) Add rows to the end of a worksheet.

1	A	B	C
2			
3			

x1	x2	x3
2	y	5
3	z	6

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_sheet()** / **range_read()** looks at the first 1000 rows of data. Increase with **guess_max**.

```
read_sheet(path, guess_max = Inf)
```

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

```
read_sheet(path, col_types = "c")
```

Set each column individually

```
# col types: skip, guess, integer, logical, character  
read_sheets(ss, col_types = "?ilc")
```

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.

Data transformation with dplyr :: CHEAT SHEET



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.

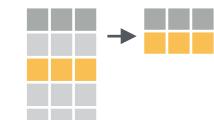
`starwars %>% rowwise() %>% mutate(film_count = length(films))`

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

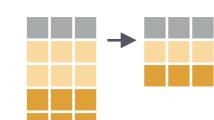
Manipulate Cases

EXTRACT CASES

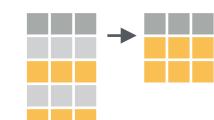
Row functions return a subset of rows as a new table.



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)`



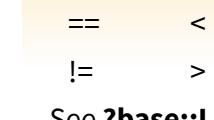
slice(.data, ..., .preserve = FALSE) Select rows by position.
`slice(mtcars, 10:15)`



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_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>	: e.g. <code>mpg:cyl</code>
<code>ends_with(match)</code>	<code>all_of(x)/any_of(x, ..., vars)</code>	- e.g. <code>-gear</code>
<code>starts_with(match)</code>	<code>matches(match)</code>	everything()

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).

vectorized function

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-NAs

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.

A	B	C	A	B
1	a	t	1	t
2	b	u	2	u
3	c	v	3	NA

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

A	B	C	A	B
1	a	t	1	a
2	b	u	2	b
3	c	v	3	c

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

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

Combine Tables

COMBINE VARIABLES

X	y	=
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	left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na")
a t 1 3	Join matching values from y to x.

A B C D	right_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na")
a t 1 3	Join matching values from x to y.

A B C D	inner_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na")
a t 1 3	Join data. Retain only rows with matches.

A B C D	full_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na")
a t 1 3	Join data. Retain all values, all rows.

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
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 B C	
a t 1	c v 3	

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).

X	y	=
A B C	A B C	
a t 1	x a t 1	

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

X	y	=
A B C	A B D	
a t 1	a t 3	

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.

X	y	=
A B C	A B C	
c v 3	NA NA	

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

X	y	=
A B C	<tibble [1x2]>	
a t 1	a t 1	

nest_join(x, y, by = NULL, copy = FALSE, keep = FALSE, name = NULL, ...) Join data, nesting matches from y in a single new data frame column.

SET OPERATIONS

A B C	c v 3
a t 1	a t 1

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

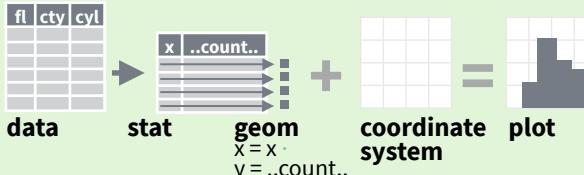
A B C	a t 1
a t 1	a t 1

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

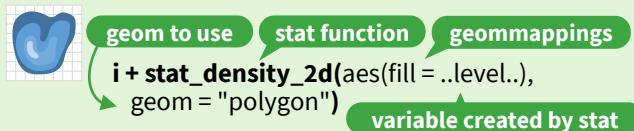
Stats

An alternative way to build a layer.

A stat builds new variables to plot (e.g., count, prop).



Visualize a stat by changing the default stat of a geom function, `geom_bar(stat="count")` or by using a stat function, `stat_count(geom="bar")`, which calls a default geom to make a layer (equivalent to a geom function). Use `..name..` syntax to map stat variables to aesthetics.



```
c + stat_bin(binwidth = 1, boundary = 10)
x, y | ..count.., ..ncount.., ..density.., ..ndensity..
c + stat_count(width = 1) x, y | ..count.., ..prop..
c + stat_density(adjust = 1, kernel = "gaussian")
x, y | ..count.., ..density.., ..scaled..
```

```
e + stat_bin_2d(bins = 30, drop = T)
x, y, fill | ..count.., ..density..
e + stat_bin_hex(bins = 30) x, y, fill | ..count.., ..density..
e + stat_density_2d(contour = TRUE, n = 100)
x, y, color, size | ..level..
e + stat_ellipse(level = 0.95, segments = 51, type = "t")
l + stat_contour(aes(z = z)) x, y, z, order | ..level..
l + stat_summary_hex(aes(z = z), bins = 30, fun = max)
x, y, z, fill | ..value..
l + stat_summary_2d(aes(z = z), bins = 30, fun = mean)
x, y, z, fill | ..value..
```

```
f + stat_boxplot(coef = 1.5)
x, y | ..lower.., ..middle.., ..upper.., ..width.., ..ymin.., ..ymax..
f + stat_ydensity(kernel = "gaussian", scale = "area") x, y
| ..density.., ..scaled.., ..count.., ..n.., ..violinwidth.., ..width..
e + stat_ecdf(n = 40) x, y | ..x.., ..y..
e + stat_quantile(quantiles = c(0.1, 0.9),
formula = y ~ log(x), method = "rq") x, y | ..quantile..
e + stat_smooth(method = "lm", formula = y ~ x, se = T,
level = 0.95) x, y | ..se.., ..x.., ..y.., ..ymin.., ..ymax..
```

```
ggplot() + xlim(-5, 5) + stat_function(fun = dnorm,
n = 20, geom = "point") x | ..x.., ..y..
```

```
ggplot() + stat_qq(aes(sample = 1:100))
x, y, sample | ..sample.., ..theoretical..
```

```
e + stat_sum() x, y, size | ..n.., ..prop..
e + stat_summary(fun.data = "mean_cl_boot")
```

```
h + stat_summary_bin(fun = "mean", geom = "bar")
```

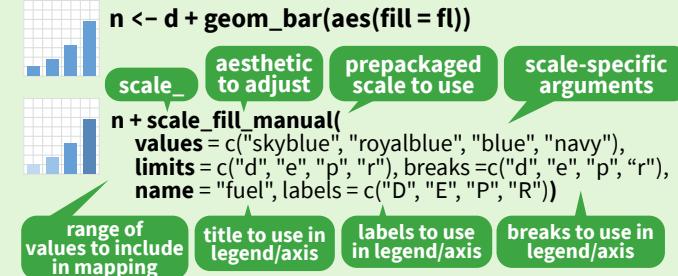
```
e + stat_identity()
```

```
e + stat_unique()
```

Scales

Override defaults with `scales` package.

Scales map data values to the visual values of an aesthetic. To change a mapping, add a new scale.



GENERAL PURPOSE SCALES

Use with most aesthetics

`scale_*_continuous()` - Map cont' values to visual ones.
`scale_*_discrete()` - Map discrete values to visual ones.
`scale_*_binned()` - Map continuous values to discrete bins.
`scale_*_identity()` - Use data values as visual ones.
`scale_*_manual(values = c())` - Map discrete values to manually chosen visual ones.
`scale_*_date(date_labels = "%m/%d"), date_breaks = "2 weeks"` - Treat data values as dates.
`scale_*_datetime()` - Treat data values as date times. Same as `scale_*_date()`. See `?strptime` for label formats.

X & Y LOCATION SCALES

Use with x or y aesthetics (x shown here)

`scale_x_log10()` - Plot x on log10 scale.
`scale_x_reverse()` - Reverse the direction of the x axis.
`scale_x_sqrt()` - Plot x on square root scale.

COLOR AND FILL SCALES (DISCRETE)

`n + scale_fill_brewer(palette = "Blues")`
For palette choices:
RColorBrewer::display.brewer.all()
`n + scale_fill_grey(start = 0.2, end = 0.8, na.value = "red")`

COLOR AND FILL SCALES (CONTINUOUS)

`o <- c + geom_dotplot(aes(fill = ..x..))`
`o + scale_fill_distiller(palette = "Blues")`
`o + scale_fill_gradient(low = "red", high = "yellow")`
`o + scale_fill_gradient2(low = "red", high = "blue", mid = "white", midpoint = 25)`
`o + scale_fill_gradientn(colors = topo.colors(6))`
Also: rainbow(), heat.colors(), terrain.colors(), cm.colors(), RColorBrewer::brewer.pal()

SHAPE AND SIZE SCALES

`p <- e + geom_point(aes(shape = fl, size = cyl))`
`p + scale_shape() + scale_size()`
`p + scale_shape_manual(values = c(3:7))`
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
`p + scale_radius(range = c(1,6))`
`p + scale_size_area(max_size = 6)`

Coordinate Systems

`r <- d + geom_bar()`

`r + coord_cartesian(xlim = c(0, 5))` - xlim, ylim
The default cartesian coordinate system.

`r + coord_fixed(ratio = 1/2)`
ratio, xlim, ylim - Cartesian coordinates with fixed aspect ratio between x and y units.

`ggplot(mpg, aes(y = fl)) + geom_bar()`
Flip cartesian coordinates by switching x and y aesthetic mappings.

`r + coord_polar(theta = "x", direction=1)`
theta, start, direction - Polar coordinates.

`r + coord_trans(y = "sqrt")` - x, y, xlim, ylim
Transformed cartesian coordinates. Set xtrans and ytrans to the name of a window function.

`pi + coord_quickmap()`
`pi + coord_map(projection = "ortho", orientation = c(41, -74, 0))` - projection, xlim, ylim
Map projections from the mapproj package (mercator (default), azequalarea, lagrange, etc.).

Position Adjustments

Position adjustments determine how to arrange geoms that would otherwise occupy the same space.

`s <- ggplot(mpg, aes(fl, fill = drv))`

`s + geom_bar(position = "dodge")`
Arrange elements side by side.

`s + geom_bar(position = "fill")`
Stack elements on top of one another, normalize height.

`e + geom_point(position = "jitter")`
Add random noise to X and Y position of each element to avoid overplotting.

`e + geom_label(position = "nudge")`
Nudge labels away from points.

`s + geom_bar(position = "stack")`
Stack elements on top of one another.

Each position adjustment can be recast as a function with manual `width` and `height` arguments:
`s + geom_bar(position = position_dodge(width = 1))`

Themes

`r + theme_bw()`
White background with grid lines.

`r + theme_gray()`
Grey background (default theme).

`r + theme_dark()`
Dark for contrast.

`r + theme_classic()`
`r + theme_light()`

`r + theme_linedraw()`
`r + theme_minimal()`

`r + theme_void()`
Empty theme.

`r + theme()` Customize aspects of the theme such as axis, legend, panel, and facet properties.
`r + ggtitle("Title") + theme(plot.title.position = "plot")`
`r + theme(panel.background = element_rect(fill = "blue"))`

Faceting

Facets divide a plot into subplots based on the values of one or more discrete variables.

`t <- ggplot(mpg, aes(cty, hwy)) + geom_point()`

`t + facet_grid(cols = vars(fl))`
Facet into columns based on fl.

`t + facet_grid(rows = vars(year))`
Facet into rows based on year.

`t + facet_grid(rows = vars(year), cols = vars(fl))`
Facet into both rows and columns.

`t + facet_wrap(vars(fl))`
Wrap facets into a rectangular layout.

Set `scales` to let axis limits vary across facets.

`t + facet_grid(rows = vars(drv), cols = vars(fl), scales = "free")`

x and y axis limits adjust to individual facets:
`"free_x"` - x axis limits adjust
`"free_y"` - y axis limits adjust

Set `labeler` to adjust facet label:

`t + facet_grid(cols = vars(fl), labeler = label_both)`

fl: c fl: d fl: e fl: p fl: r

`t + facet_grid(rows = vars(fl), labeler = label_bquote(alpha ^ .(fl)))`

$\alpha^c \quad \alpha^d \quad \alpha^e \quad \alpha^p \quad \alpha^r$

Labels and Legends

Use `labs()` to label the elements of your plot.

`t + labs(x = "New x axis label", y = "New y axis label", title = "Add a title above the plot", subtitle = "Add a subtitle below title", caption = "Add a caption below plot", alt = "Add alt text to the plot", <AES> = "New <AES> legend title")`

`t + annotate(geom = "text", x = 8, y = 9, label = "A")`
Places a geom with manually selected aesthetics.

`p + guides(x = guide_axis(n.dodge = 2))` Avoid crowded or overlapping labels with `guide_axis(n.dodge` or `angle`).

`n + guides(fill = "none")` Set legend type for each aesthetic: colorbar, legend, or none (no legend).

`n + theme(legend.position = "bottom")`
Place legend at "bottom", "top", "left", or "right".

`n + scale_fill_discrete(name = "Title", labels = c("A", "B", "C", "D", "E"))`
Set legend title and labels with a scale function.

Zooming

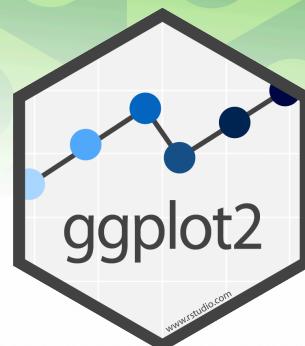
Without clipping (preferred):

`t + coord_cartesian(xlim = c(0, 100), ylim = c(10, 20))`

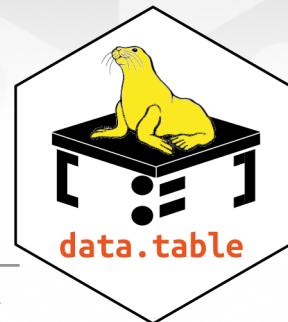
With clipping (removes unseen data points):

`t + xlim(0, 100) + ylim(10, 20)`

`t + scale_x_continuous(limits = c(0, 100)) + scale_y_continuous(limits = c(0, 100))`



Data Transformation with data.table :: CHEAT SHEET



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.

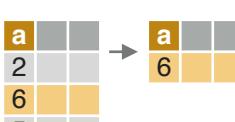
Create a data.table

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**

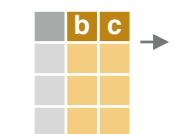
<	<=	is.na()	%in%		%like%
>	>=	!is.na()	!	&	%between%

Manipulate columns with **j**

EXTRACT

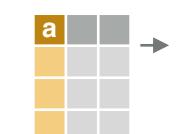


dt[, c(2)] – extract columns by number. Prefix column numbers with “-” to drop.



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

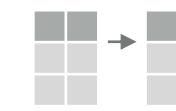
SUMMARIZE



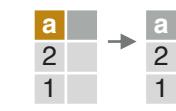
dt[, .(x = sum(a))] – create a data.table with new columns based on the summarized values of rows.

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

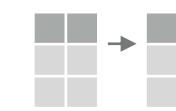
COMPUTE COLUMNS*



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

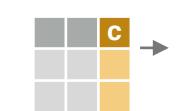


dt[a == 1, c := 1 + 2] – compute a column based on an expression but only for a subset of rows.



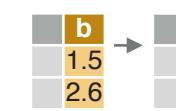
dt[, `:=` (c = 1, d = 2)] – compute multiple columns based on separate expressions.

DELETE COLUMN



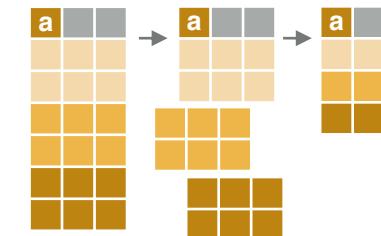
dt[, c := NULL] – delete a column.

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.

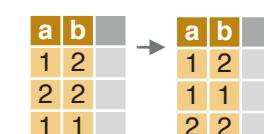
dt[, .SD[N], by = a] – extract last row of groups.

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)`”.



UNIQUE ROWS

a	b
1	2
2	2
1	2

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

`uniqueN(dt, by = c("a", "b"))` – count the number of unique rows based on columns specified in “by”.

RENAME COLUMNS

a	b
x	y

`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
1	c
2	a
3	b

x	y
3	b
2	c
1	a

a	b	x
3	b	3
2	c	2
1	a	1

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

a	b	c
1	c	7
2	a	5
3	b	6

x	y	z
3	b	4
2	c	5
1	a	8

a	b	c	x
3	b	4	3
1	c	5	2
2	a	8	NA

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

ROLLING JOIN

a	id	date
1	A	01-01-2010
2	A	01-01-2012
3	A	01-01-2014
1	B	01-01-2010
2	B	01-01-2012

b	id	date
1	A	01-01-2013
1	B	01-01-2013

a	id	date	b
2	A	01-01-2013	1
1	B	01-01-2013	1

`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

a	b

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

a	b

x	y

a	b	x	y

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

Apply function to cols.

APPLY A FUNCTION TO MULTIPLE COLUMNS

a	b
1	4
2	5
3	6

`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
1	1
2	2
3	3

`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.

RESHAPE TO WIDE FORMAT

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

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

Reshape a data.table from long to wide format.

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

RESHAPE TO LONG FORMAT

id	a	x	a	z	b	x	b	z
A	1	2	3	4				
B	1	2	3	4				

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

Sequential rows

ROW IDS

a	b
1	a
2	a
3	b

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

a	b

<tbl_r cells="2" ix="1" maxcspan="1" maxrspan="1" usedcols="2

DeclareDesign:: CHEAT SHEET

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(ATE = 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)))
```

Create, query and manipulate distributions with distr6 :: CHEAT SHEET



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

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

R6 Basics

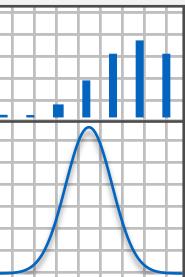
\$ All methods are called using dollar-sign notation	<code>N <- Normal\$new() N\$mean() N\$pdf(2)</code>
clone Objects are copied using the clone method	<code>N1 <- Normal\$new() N2 <- N1\$clone()</code>
Method chaining Call one method after another	<code>Normal\$new()\$pdf(2)</code>

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()
```

S3 and Piping

distr6 uses '[R6toS3](#)' so every R6 method has an S3 dispatch available.

<code>N <- Normal\$new()</code>	
<code>N\$mean()</code>	→ <code>mean(N)</code>
<code>N\$getParameterValue("mean")</code>	→ <code>getParameterValue(N, "mean")</code>

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

<code>> N <- Normal\$new()</code>	
<code>> N %>% setParameterValue(sd=2)\$getParameterValue("var")</code>	
	↓ library(magrittr)
<code>> N <- Normal\$new()</code>	
<code>> N %>% setParameterValue(sd=2) %>%</code>	
<code> getParameterValue("var")</code>	

Multivariate Distributions

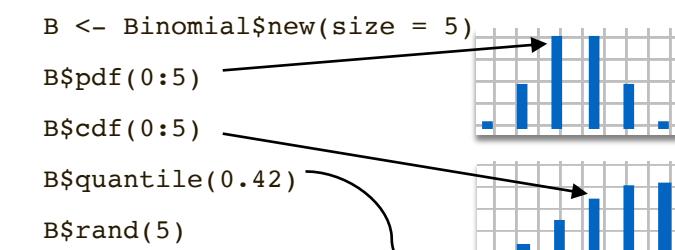
Multivariate distributions are handled just like univariate distributions, except the pdf/cdf functions take multiple arguments, as do cf and mgf where available.

<code>> MN <- MultivariateNormal\$new(mean = c(0,0,0), cov = c(3,-1,-1,-1,1,0,-1,0,1))</code>	
<code>> MN <- MultivariateNormal\$new(mean = c(0,0,0), prec = c(3,-1,-1,-1,1,0,-1,0,1))</code>	
<code>> MN\$pdf(1, 2, 3)</code>	
<code>> MN\$cdf(1, 1, 1)</code>	
Once again vectorization is available	
<code>> MN\$pdf(1:2, 2:3, 1:2)</code>	
<code>> MN\$cdf(c(0.45, 0.65), c(0.12, 0.99), c(0, 1))</code>	

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.



Create, query and manipulate distributions with distr6 :: CHEAT SHEET



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()
```

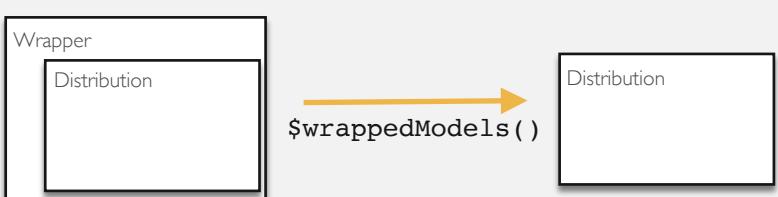
Wrappers

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	VectorDistribution
Product of two or more distributions.	Vectorizes two or more distributions.
Convolution	Addition (or subtraction) of two distributions
HuberizedDistribution	MixtureDistribution
Huberizes a distribution between limits.	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

estimatr :: CHEAT SHEET

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
```

Access Eurostat data with eurostat::cheat sheet

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")

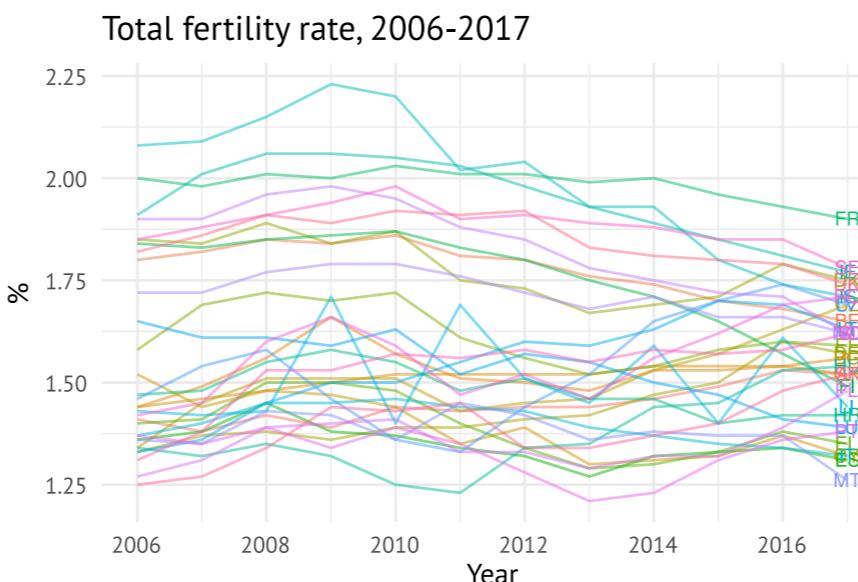
```
dat <- label_eurostat(dat)
dat[1:3,]
## indic_de      geo    time values
## <fct>        <fct>  <dbl> <dbl>
## Total fertility rate Andorra 2006  1.24
## Total fertility rate Albania 2006  1.67
## Total fertility rate Armenia 2006  1.34
```



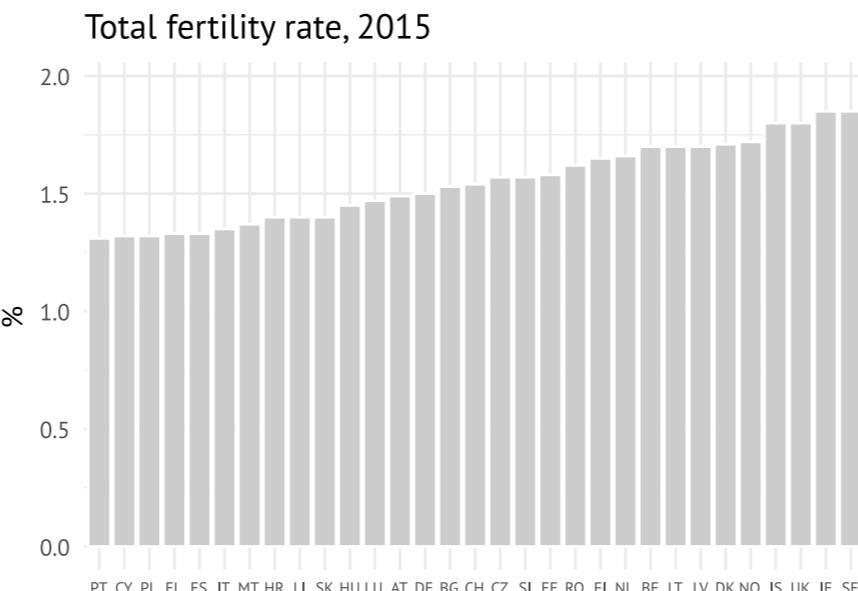
eurostat and plots

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 = "%")
```



```
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)
```



eurostat and maps

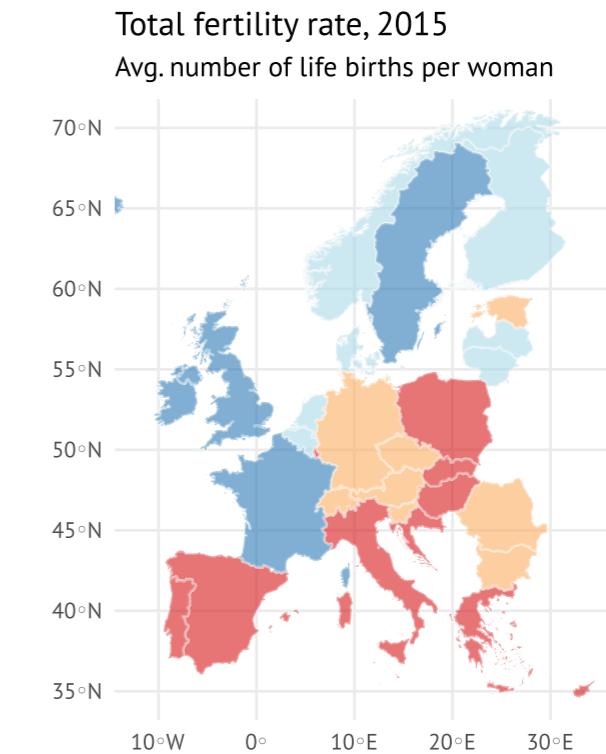
There are two function to work with geospatial data from GISCO. The **get_eurostat_geospatial()** returns spatial data as sf-object. Object can me merged with data.frames using **dplyr::*_join()**-functions. The **cut_to_classes()** is a wrapper for **cut()** - function and is used for categorizing data for maps with tidy labels.

```
mapdata <- get_eurostat_geospatial(nuts_level = 0) %>%
  right_join(dat_2015) %>%
  mutate(cat = cut_to_classes(values, n=4, decimals=1))
head(select(mapdata, geo, values, cat), 3)
## #> #> geo values      cat      geometry
## #> #> AT    1.49 1.5 ~< 1.6 MULTIPOLYGON (((15.54245 48...
```

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 = .6) +
  xlim(c(-12, 44)) + ylim(c(35, 70)) +
  labs(title = "Total fertility rate, 2015",
       subtitle = "Avg. number of life births per woman",
       fill = "%")
```



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 Package.
 Leo Lahti, Janne Huovari, Markus Kainu, and Przemysław Biecek.
 The R Journal, 9(1):385–392, 2017.



Factors withforcats :: CHEAT SHEET

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.

<code>a c b a</code>	<code>a c b a</code>	<i>Create a factor with factor()</i>
	<code>1 = a 2 = b 3 = c</code>	<code>factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x), nmax = NA)</code> Convert a vector to a factor. Also <code>as_factor()</code> . <code>f <- factor(c("a", "c", "b", "a"), levels = c("a", "b", "c"))</code>
<code>a c b a</code>	<code>a b c</code>	<i>Return its levels with levels()</i> <code>levels(x)</code> Return/set the levels of a factor. <code>levels(f); levels(f) <- c("x", "y", "z")</code>

Use unclass() to see its structure

Inspect Factors

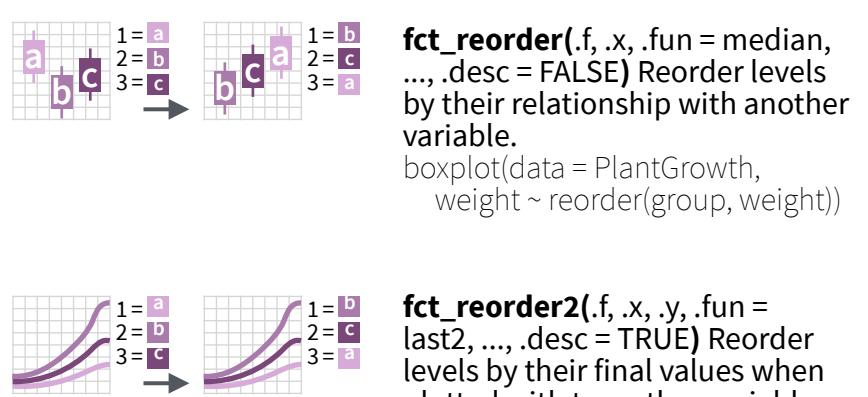
<code>a c b a</code>	<code>f n</code>	<code>fct_count(f, sort = FALSE, prop = FALSE)</code> Count the number of values with each level. <code>fct_count(f)</code>
<code>a b a</code>	<code>a 2 b 1 c 1</code>	<code>fct_match(f, lvls)</code> Check for lvls in f. <code>fct_match(f, "a")</code>
<code>a b a</code>	<code>a b 1 = a 2 = b</code>	<code>fct_unique(f)</code> Return the unique values, removing duplicates. <code>fct_unique(f)</code>

Combine Factors

<code>a c 1 = a 2 = c</code>	<code>b a 1 = a 2 = b</code>	<code>fct_c(...)</code> Combine factors with different levels. Also <code>fct_cross()</code> . <code>f1 <- factor(c("a", "c"))</code> <code>f2 <- factor(c("b", "a"))</code> <code>fct_c(f1, f2)</code>
<code>a b 1 = a 2 = b</code>	<code>a b 1 = a 2 = b 3 = c</code>	<code>fct_unify(fs, levels = lvl_union(fs))</code> Standardize levels across a list of factors. <code>fct_unify(list(f2, f1))</code>

Change the order of levels

<code>a c b a</code>	<code>a c b a</code>	<code>fct_relevel(.f, ..., after = 0L)</code> Manually reorder factor levels. <code>fct_relevel(f, c("b", "c", "a"))</code>
<code>c c a</code>	<code>c c a</code>	<code>fct_infreq(f, ordered = NA)</code> Reorder levels by the frequency in which they appear in the data (highest frequency first). Also <code>fct_inseq()</code> . <code>f3 <- factor(c("c", "c", "a"))</code> <code>fct_infreq(f3)</code>
<code>b a</code>	<code>b a</code>	<code>fct_inorder(f, ordered = NA)</code> Reorder levels by order in which they appear in the data. <code>fct_inorder(f2)</code>
<code>a b c</code>	<code>a b c</code>	<code>fct_rev(f)</code> Reverse level order. <code>f4 <- factor(c("a", "b", "c"))</code> <code>fct_rev(f4)</code>
<code>a b c</code>	<code>a b c</code>	<code>fct_shift(f)</code> Shift levels to left or right, wrapping around end. <code>fct_shift(f4)</code>
<code>a b c</code>	<code>a b c</code>	<code>fct_shuffle(f, n = 1L)</code> Randomly permute order of factor levels. <code>fct_shuffle(f4)</code>



Change the value of levels

<code>a c b a</code>	<code>v z x v</code>	<code>fct_recode(.f, ...)</code> Manually change levels. Also <code>fct_relabel()</code> which obeys purrr::map syntax to apply a function or expression to each level. <code>fct_recode(f, v = "a", x = "b", z = "c")</code> <code>fct_relabel(f, ~ paste0("x", .x))</code>
<code>a c b a</code>	<code>2 1 3 2</code>	<code>fct_anon(f, prefix = "")</code> Anonymize levels with random integers. <code>fct_anon(f)</code>
<code>a c b a</code>	<code>x c x x</code>	<code>fctCollapse(.f, ..., other_level = NULL)</code> Collapse levels into manually defined groups. <code>fct_collapse(f, x = c("a", "b"))</code>
<code>a c b a</code>	<code>a Other Other a</code>	<code>fct_lump_min(f, min, w = NULL, other_level = "Other")</code> Lumps together factors that appear fewer than min times. Also <code>fct_lump_n()</code> , <code>fct_lump_prop()</code> , and <code>fct_lump_lowfreq()</code> . <code>fct_lump_min(f, min = 2)</code>
<code>a c b a</code>	<code>a b Other Other b a</code>	<code>fct_other(f, keep, drop, other_level = "Other")</code> Replace levels with "other." <code>fct_other(f, keep = c("a", "b"))</code>

Add or drop levels

<code>a b 1 = a 2 = b 3 = x</code>	<code>a b 1 = a 2 = b</code>	<code>fct_drop(f, only)</code> Drop unused levels. <code>f5 <- factor(c("a", "b"), c("a", "b", "x"))</code> <code>f6 <- fct_drop(f5)</code>
<code>a b 1 = a 2 = b</code>	<code>a b 1 = a 2 = b 3 = x</code>	<code>fct_expand(f, ...)</code> Add levels to a factor. <code>fct_expand(f6, "x")</code>
<code>a b NA</code>	<code>a b x NA</code>	<code>fct_explicit_na(f, na_level = "(Missing)")</code> Assigns a level to NAs to ensure they appear in plots, etc. <code>fct_explicit_na(factor(c("a", "b", NA)))</code>

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_filters()

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

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

... 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...

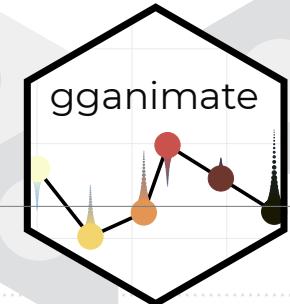
... and we're cycling between **seven** views. Seven is the number of steps in the transition, so the view is changing when the points are static, and visa versa. Views are determined by what data is in the current frame.

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/_filter/_expression	Similar to their state/layer analogs. *_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.

GWAS CATALOG	RETRIEVAL FUNCTIONS	S4 CLASSES
	get_studies()	S studies
	get_associations()	A associations
	get_variants()	V variants
	get_traits()	T traits
Search by	Example	
study_id	"CCST000858"	S A V T
association_id	"24300113"	
variant_id	"rs12752552"	
efo_id	"EFO_0005543"	
pubmed_id	"21626137"	
user_requested	TRUE	
full_pvalue_set	FALSE	
efo_uri	"http://www.ebi.ac.uk/efo/EFO_0004761"	
genomic_range	list(chromosome = "22", start = 1L, end = 15473564L)	
gene_name	"BRCA1"	
efo_trait	"lung adenocarcinoma"	
reported_trait	"Breast cancer"	
cytogenetic_band	"1p36.33"	

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>		• <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>		
• <code>snp_count</code>	• <code>study_id</code>	• <code>study_id</code>
• <code>qualifier</code>	• <code>ancestry_id</code>	• <code>ancestry_id</code>
• <code>imputed</code>	• <code>type</code>	• <code>country_name</code>
• <code>pooled</code>	• <code>number_of_individuals</code>	• <code>major_area</code>
• <code>study_design_comment</code>		• <code>region</code>
• <code>full_pvalue_set</code>	• <code>study_id</code>	
• <code>user_requested</code>	• <code>ancestry_id</code>	• <code>study_id</code>
	• <code>ancestral_group</code>	• <code>pubmed_id</code>
		• <code>publication_date</code>
		• 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>	
• <code>pvalue_exponent</code>		• <code>ensembl_id</code>
• <code>multiple.snp.haplotype</code>	• <code>association_id</code>	• <code>variant_id</code>
• <code>snp_interaction</code>	• <code>locus_id</code>	• <code>gene_name</code>
• <code>snp_type</code>	• <code>variant_id</code>	• <code>ensembl_id</code>
• <code>standard_error</code>	• <code>risk_allele</code>	• <code>entrez_id</code>
• <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>		

h2o:: CHEAT SHEET

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.

x[j] ## column J	x[i]	<- value	Value Assignment
x[i, j]	x[i, j, ...]	<- value	
x[[i]]	x[[i]]	<- value	
x\$name	x\$i	<- value	

Selection

Value Assignment

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:

h2o.asfactor, as.factor: Factor.

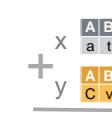
h2o.as_date, as.Date: Date.

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.

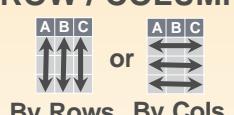
h2o.all: Given a set of logical vectors, determine if all of the values are true.

NON-GROUP_BY SUMMARIES: GENERIC

h2o.summary: Produce result summaries of the results of various model fitting functions.

Aggregations

ROW / COLUMN AGGREGATION



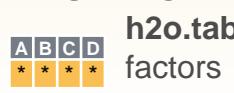
apply: Apply a function over an H2O parsed data object (an array) margins.

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.

h2o::CHEAT SHEET



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.glrn: 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.

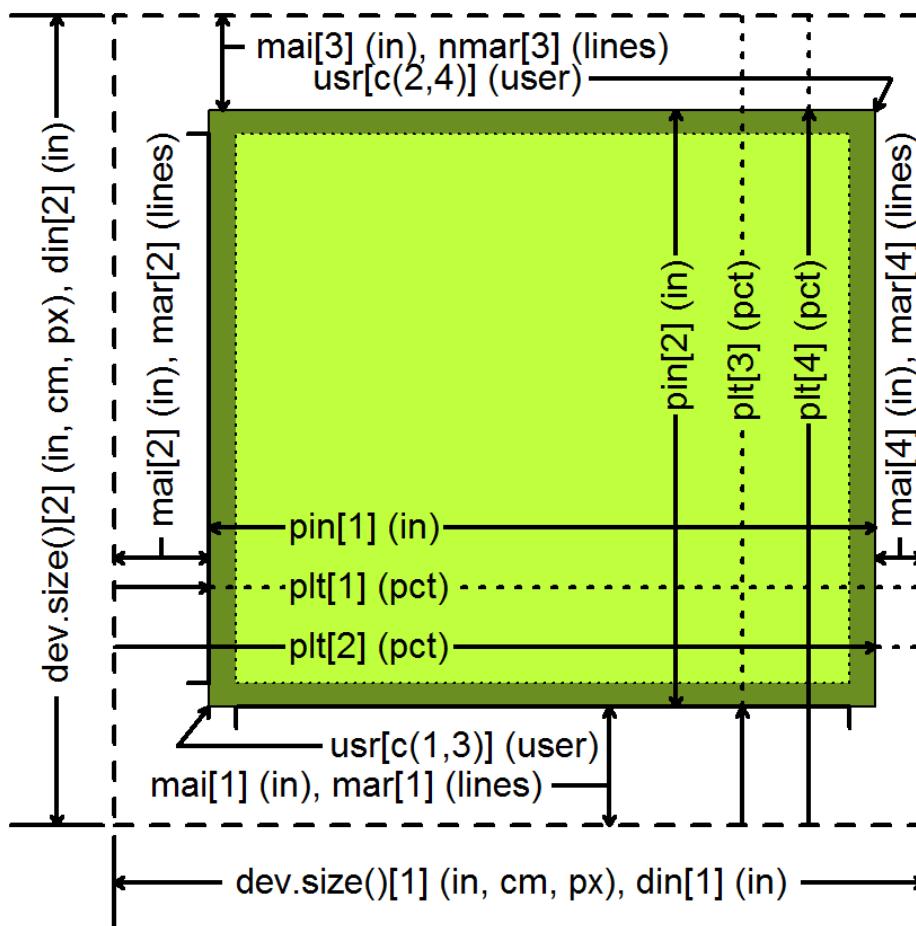


How Big is Your Graph?

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 **par** 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 ("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.

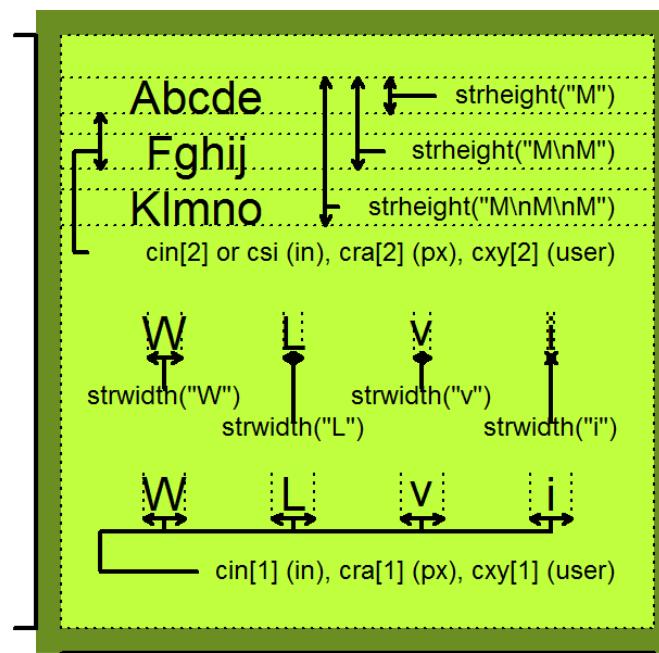
`par("cin")` (r.o.) (width, height) in inches
`par("csi")` (r.o.) height in inches
`par("cra")` (r.o.) (width, height) in pixels
`par("cxy")` (r.o.) (width, height) in xy coordinates

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

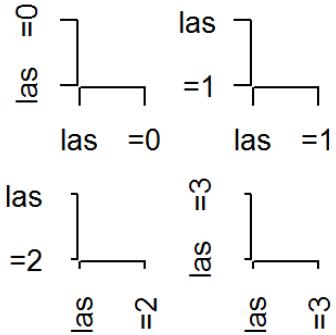
Fixing this takes a bit of trial and error.

1. Specify a larger/smaller value for the `pointsize` argument when you open your graphics device.
2. 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.
3. Use the `cex` argument to increase or decrease the relative size of your fonts.

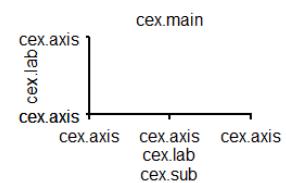
If your axes don't fit

There are several possible solutions.

1. You can assign wider margins using the `mar` or `mai` argument in `par`.
2. You can change the orientation of the axis labels with `las`. Choose among
 - a. `las=0` both axis labels parallel
 - b. `las=1` both axis labels horizontal
 - c. `las=2` both axis labels perpendicular
 - d. `las=3` both axis labels vertical.



3. change the relative size of the font
 - a. `cex.axis` for the tick mark labels.
 - b. `cex.lab` for `xlab` and `ylab`.
 - c. `cex.main` for the main title
 - d. `cex.sub` for the subtitle.





Time Series Imputation with imputeTS: : CHEAT SHEET

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_gapsizes</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")
```

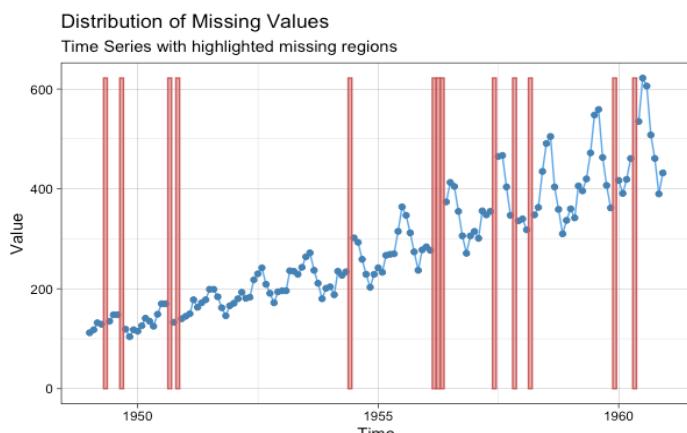
Imputation Function Your input time series Additional Parameters

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

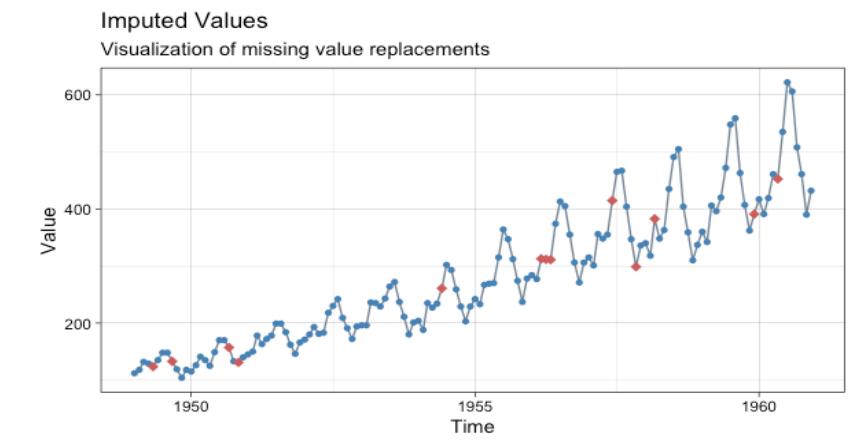
Missing Data Overview Plots

The ‘distribution’, ‘intervals’ and ‘gapsize’ plots can be used on new datasets to gain insights about missing data patterns and distribution.



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)
```

imputation visualization

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. 144 Observations / 13 NAs
<code>tsNH4</code>	NH4 concentration in a wastewater system. 3552 observations / 883 NAs
<code>tsHeating</code>	A heating systems supply temperature. 606837 observations / 57391 NAs

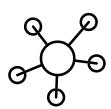
CITATION

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 = 'none',
           likelihood = 'binomial', ...)
```

- `likelihood`: Specifies the family of the prior probability distribution.

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`. A `prior` can be specified to perform Bayesian planning.

- `expectedError`: A fraction specifying the expected errors in the sample.

```
planning(materiality = 0.05,
          expectedError = 0.01,
          likelihood = 'binomial',
          confidence = 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 `algorithms`: random sampling, cell sampling, or fixed interval sampling. Sampling is done in combination with one of two sampling `units`: records or monetary units.

```
selection(population = BuildIt,
           sampleSize = 93,
           units = 'records',
           algorithm = 'interval', ...)
```

Evaluate the misstatement in the population

`jfa::evaluation()`

This function takes a data sample (using `sample`, `bookValues`, and `auditValues`) or summary statistics from a sample (using `nSumstats` and `kSumstats`) 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 = 'binomial',
            confidence = 0.95,
            nSumstats = 93,
            kSumstats = 0,
            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', ...)
```

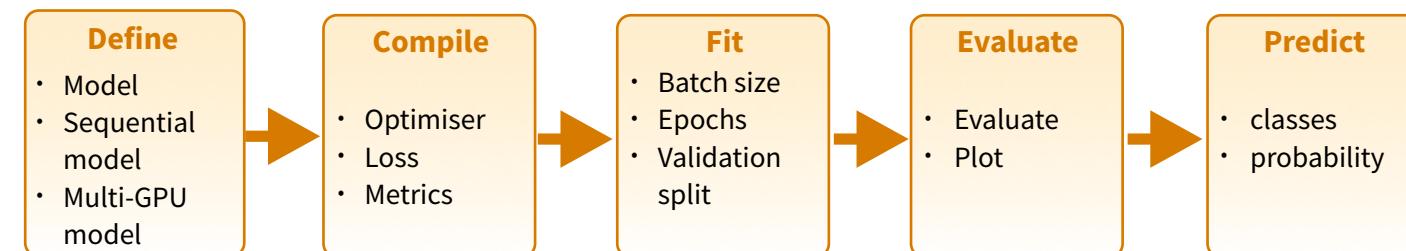
Deep Learning with Keras :: CHEAT SHEET



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 on 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

```
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)`

More layers

CONVOLUTIONAL LAYERS

	<code>layer_conv_1d()</code> 1D, e.g. temporal convolution
	<code>layer_conv_2d_transpose()</code> Transposed 2D (deconvolution)
	<code>layer_conv_2d()</code> 2D, e.g. spatial convolution over images
	<code>layer_conv_3d_transpose()</code> Transposed 3D (deconvolution) <code>layer_conv_3d()</code> 3D, e.g. spatial convolution over volumes
	<code>layer_conv_lstm_2d()</code> Convolutional LSTM
	<code>layer_separable_conv_2d()</code> Depthwise separable 2D
	<code>layer_upsampling_1d()</code> <code>layer_upsampling_2d()</code> <code>layer_upsampling_3d()</code> Upsampling layer
	<code>layer_zero_padding_1d()</code> <code>layer_zero_padding_2d()</code> <code>layer_zero_padding_3d()</code> Zero-padding layer
	<code>layer_cropping_1d()</code> <code>layer_cropping_2d()</code> <code>layer_cropping_3d()</code> Cropping layer

POOLING LAYERS

	<code>layer_max_pooling_1d()</code> <code>layer_max_pooling_2d()</code> <code>layer_max_pooling_3d()</code> Maximum pooling for 1D to 3D
	<code>layer_average_pooling_1d()</code> <code>layer_average_pooling_2d()</code> <code>layer_average_pooling_3d()</code> Average pooling for 1D to 3D
	<code>layer_global_max_pooling_1d()</code> <code>layer_global_max_pooling_2d()</code> <code>layer_global_max_pooling_3d()</code> Global maximum pooling
	<code>layer_global_average_pooling_1d()</code> <code>layer_global_average_pooling_2d()</code> <code>layer_global_average_pooling_3d()</code> Global average pooling

ACTIVATION LAYERS

	<code>layer_activation()</code> object, activation Apply an activation function to an output
	<code>layer_activation_leaky_relu()</code> Leaky version of a rectified linear unit
	<code>layer_activation_parametric_relu()</code> Parametric rectified linear unit
	<code>layer_activation_thresholded_relu()</code> Thresholded rectified linear unit
	<code>layer_activation_elu()</code> Exponential linear unit

DROPOUT LAYERS

	<code>layer_dropout()</code> Applies dropout to the input
	<code>layer_spatial_dropout_1d()</code> <code>layer_spatial_dropout_2d()</code> <code>layer_spatial_dropout_3d()</code> Spatial 1D to 3D version of dropout

RECURRENT LAYERS

	<code>layer_simple_rnn()</code> Fully-connected RNN where the output is to be fed back to input
	<code>layer_gru()</code> Gated recurrent unit - Cho et al
	<code>layer_cudnn_gru()</code> Fast GRU implementation backed by CuDNN

LOCALLY CONNECTED LAYERS

	<code>layer_locally_connected_1d()</code> <code>layer_locally_connected_2d()</code> Similar to convolution, but weights are not shared, i.e. different filters for each patch
---	--

Preprocessing

SEQUENCE PREPROCESSING

	<code>pad_sequences()</code> Pads each sequence to the same length (length of the longest sequence)
	<code>skipgrams()</code> Generates skipgram word pairs
	<code>make_sampling_table()</code> Generates word rank-based probabilistic sampling table

TEXT PREPROCESSING

	<code>text_tokenizer()</code> Text tokenization utility
	<code>fit_text_tokenizer()</code> Update tokenizer internal vocabulary
	<code>save_text_tokenizer(); load_text_tokenizer()</code> Save a text tokenizer to an external file
	<code>texts_to_sequences(); texts_to_sequences_generator()</code> Transforms each text in texts to sequence of integers
	<code>texts_to_matrix(); sequences_to_matrix()</code> Convert a list of sequences into a matrix
	<code>text_one_hot()</code> One-hot encode text to word indices

	<code>text_to_word_sequence()</code> Convert text to a sequence of words (or tokens)
	

IMAGE PREPROCESSING

	<code>image_load()</code> Loads an image into PIL format.
	<code>flow_images_from_data()</code> <code>flow_images_from_directory()</code> Generates batches of augmented/normalized data from images and labels, or a directory
	<code>image_data_generator()</code> Generate minibatches of image data with real-time data augmentation.
	<code>fit_image_data_generator()</code> Fit image data generator internal statistics to some sample data
	<code>generator_next()</code> Retrieve the next item

	<code>image_to_array(); image_array_resize()</code> <code>image_array_save()</code> 3D array representation
---	--

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



Labelled data with labelled :: CHEAT SHEET

The **labelled** package provides a set of functions and methods to handle and to manipulate labelled data, as imported with **haven** package.

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



When using labelled data?

`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.

Conversion

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



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

<code>m <- leaflet(options = leafletOptions(...))</code>	Initial geographic center of the map
<code>center</code>	Initial map zoom level
<code>zoom</code>	Minimum zoom level of the map
<code>minZoom</code>	Maximum zoom level of the map
<code>maxZoom</code>	

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 = ...)
```

```
leaflet(df) %>% addTiles() %>% addCircles()
```

library(sp) Useful functions:

SpatialPoints, SpatialLines, SpatialPolygons, ...

library(maps) 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, awesomeIcons, awesomeIconList
```

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

`addTiles()`

Default Tiles

Use `addTiles()` to add a custom map tile URL template, use `addWMSTiles()` to add WMS (Web Map Service) tiles

`providers$Stamen.Toner, CartoDB.Positron, Esri.NatGeoWorldMap`

Third-Party Tiles

`addProviderTiles()`

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:

`inputs$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

`inputs$MAPID_click` when the map background or basemap is clicked

`value -- a list with lat and lng`

`inputs$MAPID_bounds` provide the lat/long bounds of the visible map area

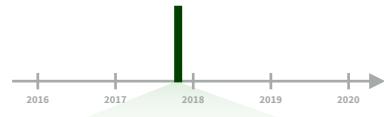
`value -- a list with north, east, south and west`

`inputs$MAPID_zoom` an integer indicates the zoom level

Dates and times with lubridate :: CHEAT SHEET



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)

1. Identify the order of the year (**y**), month (**m**), day (**d**), hour (**h**), minute (**m**) and second (**s**) elements in your data.
2. 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(zone = "") Current time in tz (defaults to system tz). now()

today(zone = "") 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"
```

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
```

GET AND SET COMPONENTS

Use an accessor function to get a component. Assign into an accessor function to change a component in place.

```
d ## "2017-11-28"  
day(d) ## 28  
day(d) <- 1  
d ## "2017-11-01"
```

2018-01-31 11:59:59

date(x) Date component. date(dt)

2018-01-31 11:59:59

year(x) Year. year(dt)
isoyear(x) The ISO 8601 year.
epiyear(x) Epidemiological year.

2018-01-31 11:59:59

month(x, label, abbr) Month. month(dt)

2018-01-31 11:59:59

day(x) Day of month. day(dt)
wday(x, label, abbr) Day of week.
qday(x) Day of quarter.

2018-01-31 11:59:59

hour(x) Hour. hour(dt)

2018-01-31 11:59:59

minute(x) Minutes. minute(dt)

2018-01-31 11:59:59

second(x) Seconds. second(dt)

2018-01-31 11:59:59 UTC

tz(x) Time zone. tz(dt)

2018-01-31 11:59:59

week(x) Week of the year. week(dt)
isoweek() ISO 8601 week.
epiweek() Epidemiological week.

2018-01-31 11:59:59

quarter(x) Quarter. quarter(dt)

2018-01-31 11:59:59

semester(x, with_year = FALSE) Semester. semester(dt)

2018-01-31 11:59:59

am(x) Is it in the am? am(dt)
pm(x) Is it in the pm? pm(dt)

2018-01-31 11:59:59

dst(x) Is it daylight savings? dst(dt)

2018-01-31 11:59:59

leap_year(x) Is it a leap year?
leap_year(dt)

2018-01-31 11:59:59

update(object, ..., simple = FALSE)
update(dt, mday = 2, hour = 1)



Round Date-times



floor_date(x, unit = "second")
Round down to nearest unit.
floor_date(dt, unit = "month")

round_date(x, unit = "second")
Round to nearest unit.
round_date(dt, unit = "month")

ceiling_date(x, unit = "second", change_on_boundary = NULL)
Round up to nearest unit.
ceiling_date(dt, unit = "month")

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()**.

1. Derive a template, create a function
sf <- stamp("Created Sunday, Jan 17, 1999 3:34")

2. 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.

5:00 Mountain 6:00 Central
4:00 Pacific 7:00 Eastern

PT MT CT ET
7:00 Pacific 7:00 Eastern

7:00 Mountain 7:00 Central

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

Math with date-times relies on the **timeline**, which behaves inconsistently. Consider how the timeline behaves during:

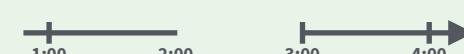
A normal day

```
nor <- ymd_hms("2018-01-01 01:30:00",tz="US/Eastern")
```



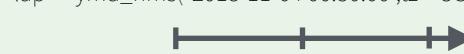
The start of daylight savings (spring forward)

```
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")
```



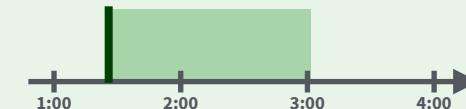
Leap years and leap seconds

```
leap <- ymd("2019-03-01")
```

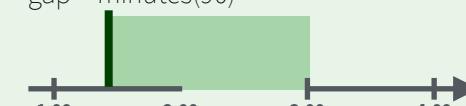


Periods track changes in clock times, which ignore time line irregularities.

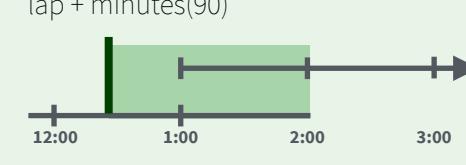
nor + minutes(90)



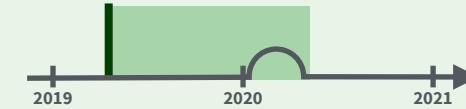
gap + minutes(90)



lap + minutes(90)

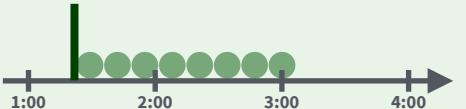


leap + years(1)



Durations track the passage of physical time, which deviates from clock time when irregularities occur.

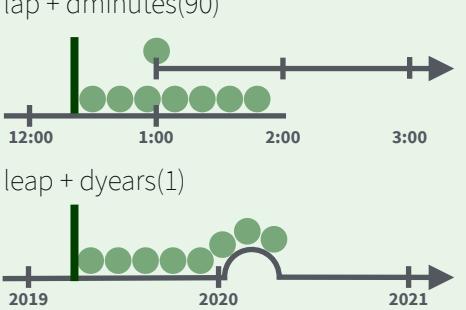
nor + dminutes(90)



gap + dminutes(90)



lap + dminutes(90)



leap + dyears(1)



Intervals represent specific intervals of the timeline, bounded by start and end date-times.

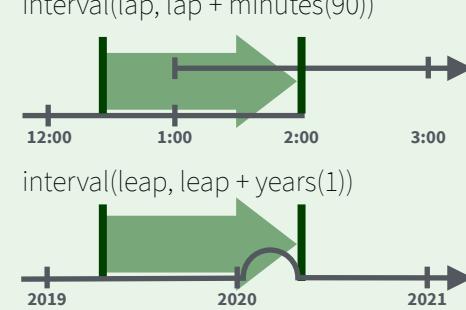
interval(nor, nor + minutes(90))



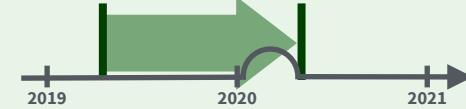
interval(gap, gap + minutes(90))



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(20180131)
```

```
jan31 + months(1)
```

```
## "NA"
```

%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)
```

```
p
```

```
"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.

Diftimes 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 × 10⁻³ seconds.

dmicroseconds(x = 1) x × 10⁻⁶ seconds.

dnanoseconds(x = 1) x × 10⁻⁹ seconds.

dpicoseconds(x = 1) x × 10⁻¹² 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.

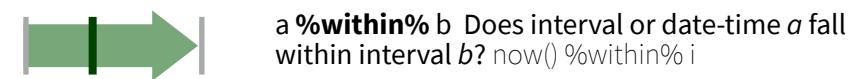
Make an interval with **interval()** or %--%, e.g.

```
i <- interval(ymd("2017-01-01"), d)
```

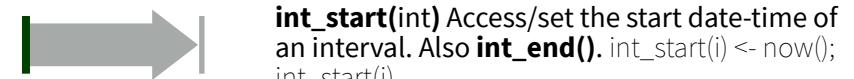
```
## 2017-01-01 UTC--2017-11-28 UTC
```

```
j <- d %--% ymd("2017-12-31")
```

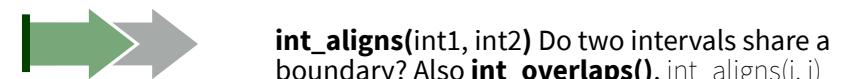
```
## 2017-11-28 UTC--2017-12-31 UTC
```



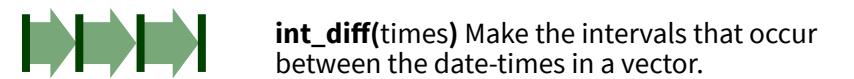
a %within% b Does interval or date-time a fall within interval b? now() %within% i



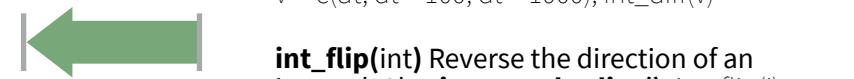
int_start(int) Access/set the start date-time of an interval. Also **int_end**(). int_start(i) <- now(); int_start(i)



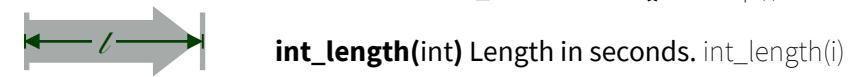
int_aligns(int1, int2) Do two intervals share a boundary? Also **int_overlaps**(). int_aligns(i, j)



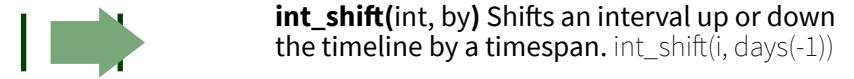
int_diff(times) Make the intervals that occur between the date-times in a vector. v <- c(dt, dt + 100, dt + 1000); int_diff(v)



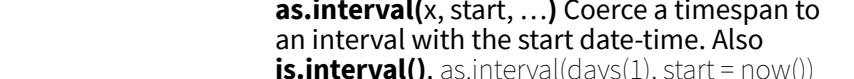
int_flip(int) Reverse the direction of an interval. Also **int_standardize**(). int_flip(i)



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 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.

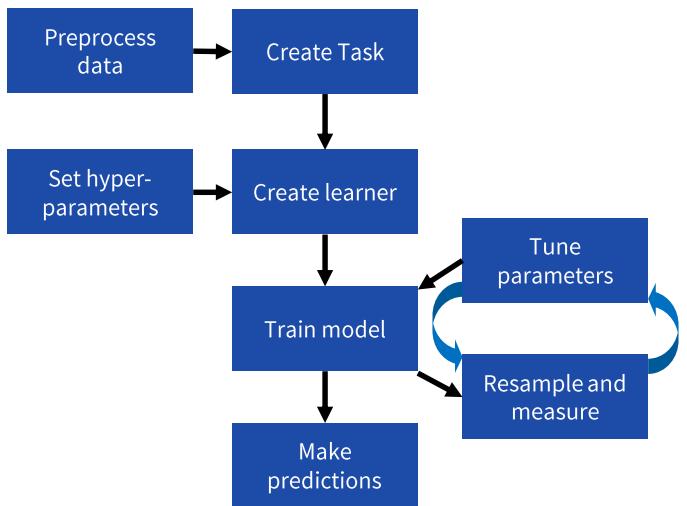
Predictions are the results of applying a model to either new data or the original training data.

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

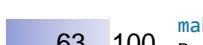
`summarizeColumns(obj=)` where `obj` is a data.frame or task.
Provides type, NA, and distributional data about each column

See also `capLargeValues` `dropFeatures` `removeConstantFeatures` `summarizeLevels`

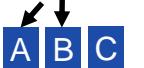
Creating a task



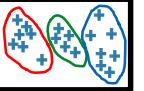
`makeClassifTask(data=, target=)`
Classification of a target variable, with optional positive class `positive`



`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

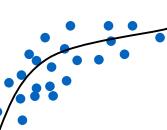
Setting hyperparameters

`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

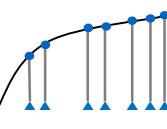


`getParamSet(learner=)`
Show the possible universe of parameters for your learner; can take a learner directly, or a text string such as `"classif.qda"`

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` `ber` `brier[,scaled]` `f1` `fdr` `fnr` `fpr` `gmean` `multiclass[,au1]` `aunp` `aunu` `brier` `npv` `ppv` `qsr` `ssr` `tn` `tnr` `tp` `tpr` `wkappa`
- `regr` `rsq` `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` `timelpredict` `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.

mlr includes 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()`, `bootstrap00B()`, `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=10)` 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)

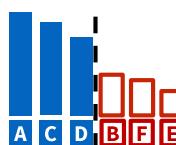
\*Can also use `classes` and `dummy.classes` in place of `cols`

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` 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

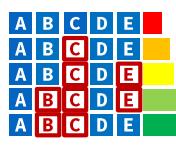
## Feature filtering



`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`

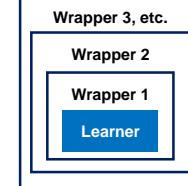
## Benchmarking

- `plotBMRBoxplots(bmr=)` Distribution of performances
- `plotBMRSummary(bmr=)` Scatterplot of avg. performances
- `plotBMRanksAsBarChart(bmr=)` Rank learners in bar plot

## Other

- `generateCritDifferencesData(bmr=, measure=, 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

# Intro stats with mosaic

(lattice version)

## Essential R syntax

Names in R are *case sensitive*

Function and arguments

`rflip(10)`

Optional arguments

`rflip(10, prob = 0.8)`

Assignment

`x <- rflip(10, prob = 0.8)`

Getting help on any function

`help(mean)`

## Loading packages

`library(mosaic)`

## Arithmetic operations

`+ - * /` basic operations

`^` exponentiation

`( )` grouping

`sqrt(x)` square root

`abs(x)` absolute value

`log10(x)` logarithm, base 10

`log(x)` natural logarithm, base  $e$

`exp(x)` exponential function  $e^x$

`factorial(k)`  $k! = k(k-1) \dots 1$

## Logical operators

`==` is equal to (note double equal sign)

`!=` is not equal to

`<` is less than

`<=` is less than or equal to

`>` is greater than

`>=` is greater than or equal to

`&` **A & B** is TRUE if both **A** and **B** are TRUE

`|` **A | B** is TRUE if one or both of **A** and **B** are TRUE

`%in%` includes; for example

`"C" %in% c("A", "B")` is FALSE

## Formula interface

Use for graphics, statistics, inference, and modeling operations.

`goal(y ~ x, data = mydata)`  
Read as “Calculate **goal** for **y** using **mydata** “broken down by” **x**, or “modeled by” **x**.

`mean(age ~ sex, data = HELPrc)`

For graphics:

`goal(y ~ x | z, groups = w, data = mydata)`

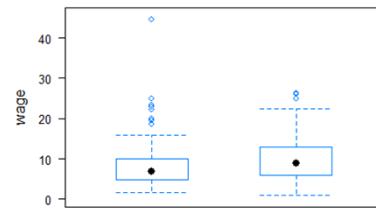
**y** : y-axis variable (*optional*)

**x** : x-axis variable (*required*)

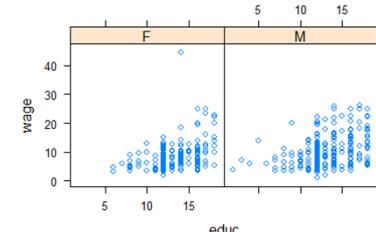
**z** : panel-by variable (*optional*)

**w** : color-by variable (*optional*)

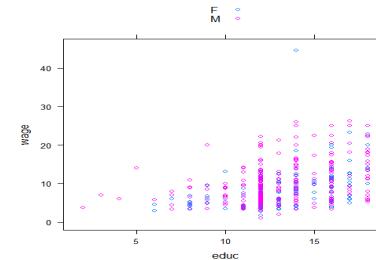
`bwplot(wage ~ sex, data = CPS85)`



`xyplot(wage ~ educ | sex, data = CPS85)`



`xyplot(wage ~ educ, groups = sex, data = CPS85, auto.key = TRUE)`



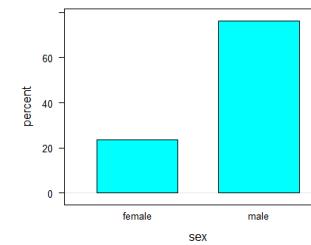
## One categorical variable

Counts by category

`tally(~ sex, data = HELPrc)`

Percentages by category

`tally(~ sex, format = "percent", data = HELPrc)`  
`bargraph(~ sex, type = "percent", data = HELPrc)`



Tests and confidence intervals

Exact test

`result1 <- binom.test(~ (homeless == "homeless"), data = HELPrc)`

Approximate test (large samples)

`result2 <- prop.test(~ (homeless == "homeless"), data = HELPrc)`

Extract confidence intervals and p-values

`confint(result1)`  
`pval(result2)`

## Examining data

Print short summary of all variables

`inspect(HELPrc)`

Number of rows and columns

`dim(HELPrc)`

`nrow(HELPrc)`

`ncol(HELPrc)`

Print first rows or last rows

`head(KidsFeet)`  
`tail(KidsFeet, 10)`

Names of variables

`names(HELPrc)`

## One quantitative variable

Make output more readable

`options(digits = 3)`

Compute summary statistics

`mean(~ cesd, data = HELPrc)`

Other summary statistics work similarly

`median()` `iqr()` `max()` `min()`

`fivenum()` `sd()` `var()` `sum()`

Table of summary statistics

`favstats(~ cesd, data = HELPrc)`

Summary statistics by group

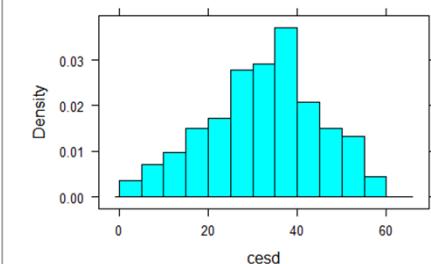
`favstats(cesd ~ sex, data = HELPrc)`

Quantiles

`quantile(~ cesd, data = HELPrc, prob = c(0.25, 0.5, 0.8))`

Histogram

`histogram(~ cesd, width = 5, center = 2.5, data = HELPrc)`



Normal probability plot

`qqmath(~ cesd, dist = "qnorm", data = HELPrc)`

Density plot

`densityplot(~ cesd, data = HELPrc)`

Dot plot

`dotPlot(~ cesd, data = HELPrc)`

One-sample t-test

`result <- t.test(~ cesd, mu = 34, data = HELPrc)`

Extract confidence intervals and p-values

`confint(result)`  
`pval(result)`

## Two categorical variables

Contingency table with margins

```
tally(~ substance + sex,
 margins = TRUE,
 data = HELPrc)
```

Percentages by column

```
tally(~ sex | substance,
 format = "percent",
 data = HELPrc)
```

Mosaic plot

```
mosaicplot(~ substance + sex,
 color = TRUE, data = HELPrc)
```



Chi-square test

```
xchisq.test(~ substance + sex,
 data = HELPrc,
 correct = FALSE)
```

## Distributions

Normal distribution function

```
pnorm(13, mean = 10, sd = 2)
```

Normal distribution function with graph

```
xpnorm(1.645, mean = 0, sd = 1)
```

Normal distribution quantiles

```
qnorm(0.95) # mean = 0, sd = 1
```

Normal distribution quantiles with graph

```
xqnorm(0.85, mean = 10, sd = 2)
```

Binomial density function ("size" means  $n$ )

```
dbinom(5, size = 8, prob = 0.65)
```

Binomial distribution function

```
pbinom(5, size = 8, prob = 0.65)
```

Central portion of distribution

```
cdist("norm", 0.95)
```

```
cdist("t", c(0.90, 0.99), df = 5)
```

Plotting distributions

```
plotDist("binom", size = 8,
 prob = 0.65, xlim = c(-1, 9))
```

```
plotDist("norm", mean = 10,
 sd = 2)
```

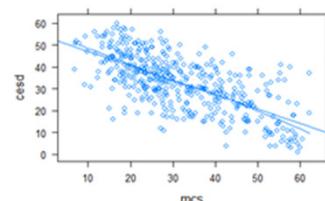
## Two quantitative variables

Correlation coefficient

```
cor(cesd ~ mcs, data = HELPrc)
```

Scatterplot with regression line and smooth

```
xyplot(cesd ~ mcs,
 type = c("p", "r", "smooth"),
 data = HELPrc)
```



Simple linear regression

```
cesdmodel <- lm(cesd ~ mcs,
 data = HELPrc)
msummary(cesdmodel)
```

Prediction

```
lmfunction <- makeFun(cesdmodel)
lmfunction(mcs = 35)
```

Extract useful quantities

```
anova(cesdmodel)
coef(cesdmodel)
confint(cesdmodel)
rsquared(cesdmodel)
```

Diagnostics; plot residuals

```
histogram(~resid(cesdmodel),
 density = TRUE)
qqmath(~resid(cesdmodel))
```

Diagnostics; plot residuals vs. fitted

```
xyplot(resid(cesdmodel) ~
 fitted(cesdmodel),
 type = c("p", "smooth", "r"))
```

## Categorical response, quantitative predictor

Logistic regression

```
logit_mod <-
 glm(homeless ~ age + female,
 family = binomial, data = HELPrc)
msummary(logitmod)
```

```
exp(coef(logit_mod))
exp(confint(logit_mod))
```

## Data management

From dplyr package

Drop or reorder variables

```
select()
```

Create new variables from existing ones

```
mutate()
```

Retain specific rows from data

```
filter()
```

Sort data rows

```
arrange()
```

Compute summary statistics by group

```
group_by()
```

```
summarize()
```

Merge data tables

```
left_join()
```

```
inner_join()
```

## Importing data

Import file from computer or URL

```
MustangPrice <-
 read.file("C:/MustangPrice.csv")
NOTE: R uses forward slashes!
Dome <-
 read.file("http://www.mosaic-
web.org/go/datasets/Dome.csv")
```

## Randomization and simulation

Fix random number sequence

```
set.seed(42)
```

Tossing coins

```
rflip(10) # default prob is 0.5
```

Do something repeatedly

```
do(5) * rflip(10, prob = 0.75)
```

Draw a simple random sample

```
sample(LETTERS, 10)
```

```
deal(Cards, 5) # poker hand
```

Resample with replacement

```
Small <- sample(KidsFeet, 10)
```

```
resample(�Small)
```

Random permutation (shuffling)

```
shuffle(Cards)
```

Random values from distributions

```
rbinom(5, size = 10, prob = 0.7)
```

```
rnorm(5, mean = 10, sd = 2)
```

## Quantitative response, categorical predictor

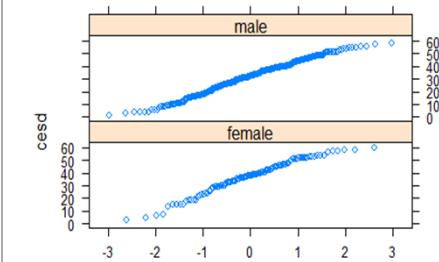
Two-level predictor: two-sample  $t$  test

Numeric summaries

```
favstats(~cesd | sex,
 data = HELPrc)
```

Comparative normal probability plot

```
qqmath(~cesd | sex, data = HELPrc,
 layout = c(1, 2)) # also bwplot
```



Dotplot for smaller samples

```
xyplot(sex ~ length, alpha = 0.6,
 cex = 1.4, data = KidsFeet)
```

Two-sample  $t$ -test and confidence interval

```
result <- t.test(cesd ~ sex,
 var.equal = FALSE, data = HELPrc)
confint(result)
```

More than two levels: Analysis of variance

Numeric summaries

```
favstats(cesd ~ substance,
 data = HELPrc)
```

Graphic summaries

```
bwplot(cesd ~ substance, pch = "|",
 data = HELPrc)
```

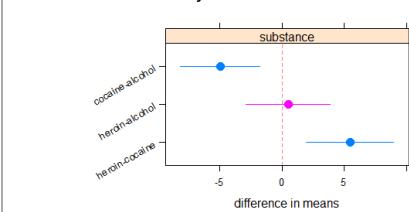
Fitt and summarize model

```
modsubstance <- lm(cesd ~ substance,
 data = HELPrc)
anova(modsubstance)
```

Which differences are significant?

```
pairwise <- TukeyHSD(modsubstance)
mplot(pairwise)
```

95% family-wise confidence level



# nardl Package

An R package to estimate the nonlinear cointegrating autoregressive distributed lag model

## 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:**

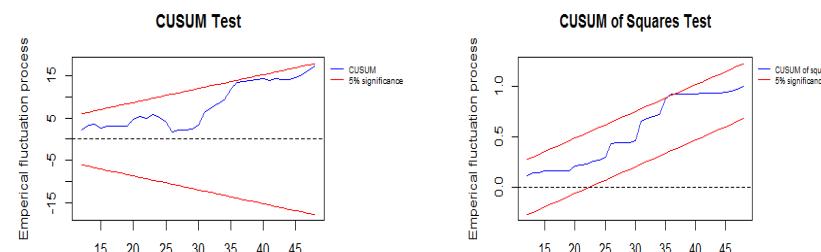
- $y \sim x | z_1 + z_2 \dots$
- $y$  the dependent variable
- $x$  the decomposed variable (this package version can't assume more than one decomposed variable)
- $z_1 + z_2 + \dots$  independent variables

- Data** is the dataframe
- p** number of lags of the dependent variable
- q** number of lags of the independent variables
- ic** : c("aic", "bic", "ll", "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)
```



## Cointegration bounds test

```
pssbounds(obs, fstat, tstat = NULL, case, k)
```

**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)
```

## Dynamic multipliers plot

Dynamic multiplier plot

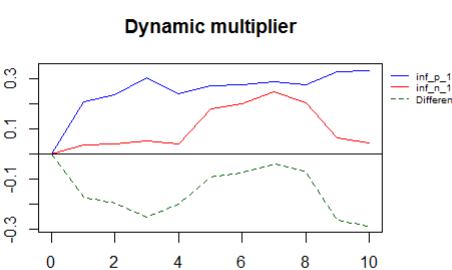
```
plotmplier(model, np, k, h)
```

**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



## 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.**

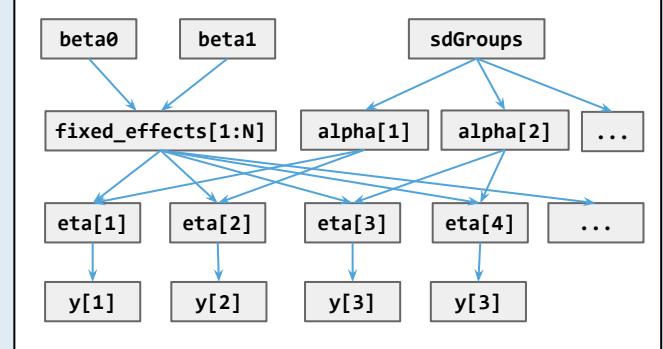
`model$calculate(nodes)`  
returns sum of log probability densities.

`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** `myModel$plot()`



**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$isNullData(nodes)`  
`model$Discrete(nodes)`  
`model$Multivariate(nodes)`  
`model$Binary(nodes)`  
`model$EndNode(nodes)`  
`model$Truncated(nodes)`

# nimble distributions and functions: : CHEAT SHEET



## Declarations

**STOCHASTIC**  
 $x \sim ddist(args)$

**DETERMINISTIC**  
 $z \leftarrow fn(args)$

**TRUNCATED STOCHASTIC**  
 $x \sim T(ddist(args), min, max)$

**CENSORED STOCHASTIC**  
 $seg \sim dinterval(t, c[1:nSegments])$   
 $t \sim ddist(args)$

**CONSTRAINT**  
 $one \sim dconstraint(condition)$

## 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



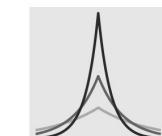
#### BETA

$y \sim dbeta([shape1, shape2 | mean, sd])$   
 $shape1 = mean^2 * (1 - mean) / sd^2 - mean$   
 $shape2 = mean * (1 - mean)^2 / sd^2 + mean - 1$



#### CHI-SQUARE

$y \sim dchisq(df)$



#### DOUBLE EXPONENTIAL (LAPLACE)

$y \sim ddexp(location, [scale|rate|var])$   
 $scale = 1/rate$   
 $scale = sqrt(var/2)$



#### EXPONENTIAL

$y \sim dexp([rate|scale])$   
 $rate = 1/scale$



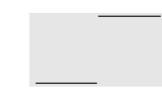
#### FLAT (improper)

$y \sim dflat()$



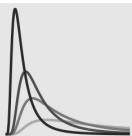
#### GAMMA

$y \sim dgamma([shape, [rate|scale] | [mean, sd]])$   
 $scale = 1/rate$   
 $shape = mean^2 / sd^2$   
 $scale = sd^2 / mean$



#### HALF FLAT (improper)

$y \sim dhalfflat()$



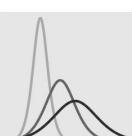
#### INVERSE GAMMA

$y \sim dinvgamma(shape, [rate|scale])$   
 $rate = 1/scale$



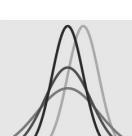
#### LOGISTIC

$y \sim dlogis(location, [rate|scale])$   
 $scale = 1/rate$



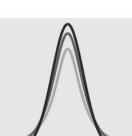
#### LOG-NORMAL

$y \sim dlnorm(meanlog, [taulog|sdlog|varlog])$   
 $sdlog = 1/sqrt(taulog)$   
 $sdlog = sqrt(varlog)$



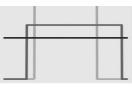
#### NORMAL

$y \sim dnorm(mean, [tau|sd|var])$   
 $sd = 1/sqrt(tau)$   
 $sd = sqrt(var)$



#### STUDENT T

$y \sim dt(mu, [tau|sigma|sigma2], df)$   
 $sigma = 1/sqrt(tau)$   
 $sigma = sqrt(sigma2)$



#### UNIFORM

$y \sim dunif(min, max)$



#### WEIBULL

$y \sim dweib(shape, [lambda|scale|rate])$   
 $scale = lambda^{-1/shape}$   
 $scale = 1/rate$

## DISTRIBUTION NAME

$y \sim ddist([default|alternative])$   
 canonical = fn(provided)

Lifted nodes are inserted when non-canonical parameters are used. Default parameters are not necessarily canonical.

## Discrete



#### BERNOULLI

$y \sim dbern(prob)$



#### BINOMIAL

$y \sim dbinom(prob, size)$



#### CATEGORICAL

$y \sim dcat(prob)$



#### NEGATIVE BINOMIAL

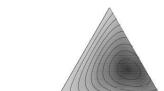
$y \sim dnegbin(prob, size)$



#### POISSON

$y \sim dpois(lambda)$

## Multivariate distributions



#### DIRICHLET

$y[] \sim ddirch(alpha[])$



#### MULTINOMIAL

$y[] \sim dmulti(prob[], size)$



#### MULTIVARIATE NORMAL

$y[] \sim dmnorm(mean[], [prec[,] | cov[,] | cholesky[,], prec_param])$



#### MULTIVARIATE STUDENT T

$y[] \sim dmvt(mu[], [prec[,] | scale[,] | cholesky[,], df, prec_param])$

cholesky = chol(prec) : prec\_param=1  
 cholesky = chol(cov) : prec\_param=0 for dmnorm  
 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

$y[,] \sim dwish([R[,] | S[,] | cholesky[,], df, scale_param])$

#### INVERSE WISHART

$y[,] \sim dinvwish([S[,] | R[,] | cholesky[,], df, scale_param])$

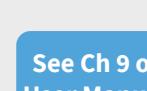
cholesky = chol(R): scale\_param=0  
 cholesky = chol(S): scale\_param=1  
 cholesky is chol(S) when scale\_param=1,  
 chol(R) when scale\_param=0

## Distributions for spatial models



#### CONDITIONAL AUTOREGRESSIVE intrinsic (improper)

$y[] \sim dcar_normal(adj[], weights[], num[], tau, c, zero_mean)$



#### proper

See Ch 9 of User Manual  
 $y[] \sim dcar_proper(mu[], C[], adj[], num[], M[], tau, gamma)$

## Bayesian nonparametric distributions



#### CHINESE RESTAURANT PROCESS

$y[] \sim dCRP(conc, size)$   
 conc = concentration parameter



#### STICK BREAKING PROCESS

$y[] \sim stick_breaking(z[])$   
 $z = \text{vector of breaking points}$

# 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**.

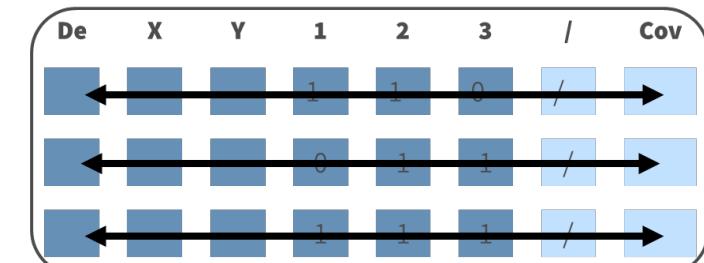
### 1.1 edf: encounter data file.

- 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)
  - X (required, UTM)
  - Y (required, UTM)
  - Binary trap operation data for malfunctions, rotations (required if problems were found;
  - 1, 2, 3, ... n)
  - Separator (e.g., /)
  - Trap level covariates (different column per covariate)



\*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.

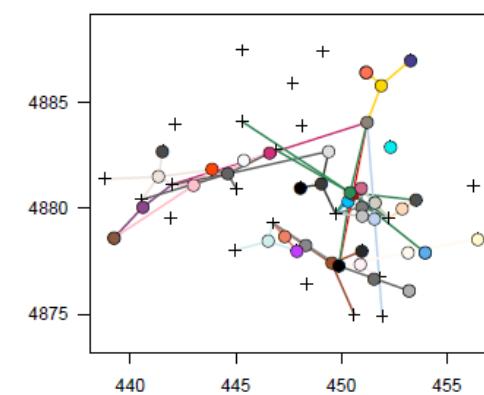
## 1.5 Summary of scrFrame

**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

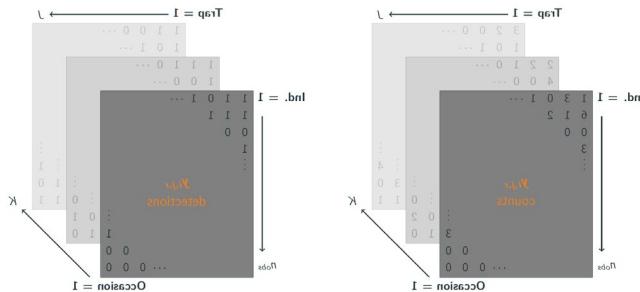
`plot(sf)` #y and x are UTM



# 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

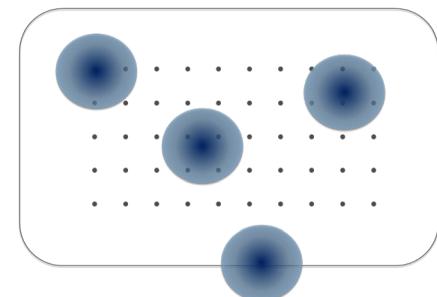
```
sf$ occasions
sf$mmdm
sf$mdm
```

## 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).



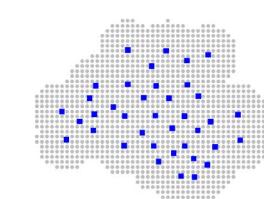
### 2.1. make.ssDF():

- Remember that  $\frac{1}{2} \text{mmdm} \sim \sigma$
- Extracts covariates and removes non habitat

```
ss <- make.ssDF(scrFrame,
buffer, #~3 to 4 σ around traps
res) # $\leq \hat{\sigma}$
```

### 2.2. Plot the state space

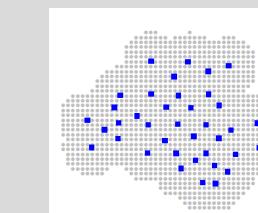
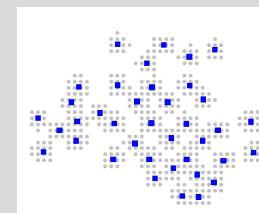
- Plot state space  
`plot(ss)`
- Plot state space & traps  
`plot(ss, sf)`



### Vary the buffer and/or resolution

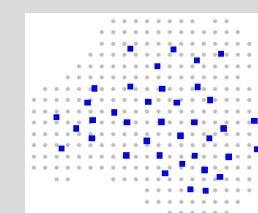
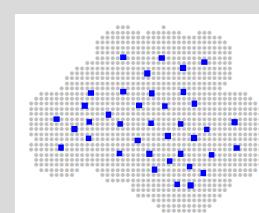
- ↗ Varying buffer, fixed resolution

```
make.ssDF(sf,
buffer = 1,
res = 0.5)
```



- ↖ Fixed buffer, varying resolution

```
make.ssDF(sf,
buffer = 3,
res = 0.1)
```



## 3. Fit the model

### 3.1. Single-session model: Fit the model with oSCR.fit():

```
sf <- data$scrFrame
mod <- oSCR.fit(model,
scrFrame, #sf
ssDF, ...)
```

- See pg. 3 for null model and multi-session models.

**model** is a list with 3 basic formulations:

```
list(D ~ 1, p0 ~ 1, sig ~ 1)
```

| Variation in... |                              |
|-----------------|------------------------------|
|                 |                              |
| <b>D</b>        | pixel density                |
| <b>p0</b>       | baseline encounter prob/rate |
| <b>sig</b>      | sigma ( $\sigma$ )           |

### 3.2. Backtransform to the real scale

```
get.real(model,
newdata,
d.factor,
type)
```

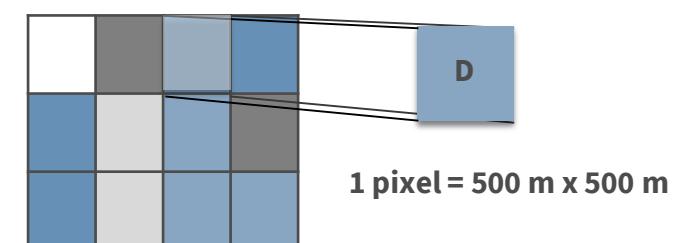
|                 |                                                               |
|-----------------|---------------------------------------------------------------|
| <b>model</b>    | fitted model                                                  |
| <b>newdata</b>  | Optional new data object for predictions                      |
| <b>d.factor</b> | optional scale the estimates to a different resolution        |
| <b>type</b>     | density ("dens"), detection probability ("det"), sigma("sig") |

**"dens"**  
Sex-specific estimates of density, and the density estimates are per pixel.

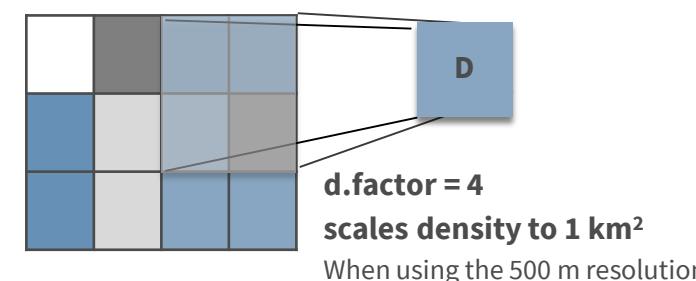
**"det"**  
Estimate of detection at distance from activity center = 0.

**"sig"**  
Estimates of the spatial scale of detection.

### d.factor



1 pixel = 500 m x 500 m



**d.factor = 4**  
scales density to 1 km²  
When using the 500 m resolution

# 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

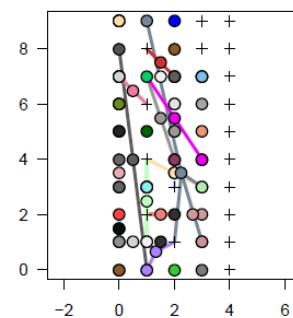
|                  | S1   | S2   | S3   | S4   |
|------------------|------|------|------|------|
| n individuals    | 77   | 60   | 108  | 54   |
| n traps          | 50   | 50   | 50   | 50   |
| n occasions      | 7    | 5    | 6    | 4    |
|                  | S1   | S2   | S3   | S4   |
| avg caps         | 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

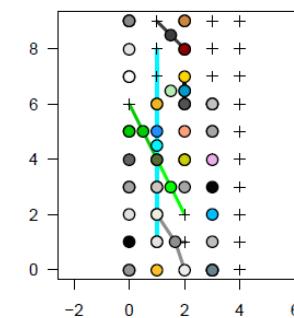
- Use `plot(sf)` to plot a spatial capture per session

```
par(mfrow=c(1,n)) # n = sessions
plot(sf) # plot all sessions
```

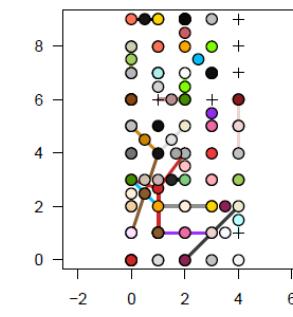
Session 1



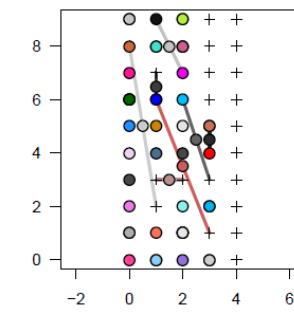
Session 2



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 ( $\sigma$ )**

| Model     | Algebra                                                  | In <code>oSCR.fit</code>   |
|-----------|----------------------------------------------------------|----------------------------|
| Density   | $\log(D_{(s_i)}) = \beta_0$                              | <code>D ~ 1</code>         |
| Density   | $\log(D_{(s_i)}) = \beta_0 + \beta_{1(g)} Session_g$     | <code>D ~ session</code>   |
| Detection | $\text{logit}(p_0) = \alpha_0$                           | <code>p0 ~ 1</code>        |
| Detection | $\text{logit}(p_0) = \alpha_0 + \alpha_{1(g)} Session_g$ | <code>p0 ~ session</code>  |
| Space use | $\log(\sigma) = \gamma_0$                                | <code>sig ~ 1</code>       |
| Space use | $\log(\sigma) = \gamma_0 + \gamma_{1(g)} Session_g$      | <code>sig ~ session</code> |

- 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)
```

Get an Overview with overviewR: : CHEAT SHEET



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
		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

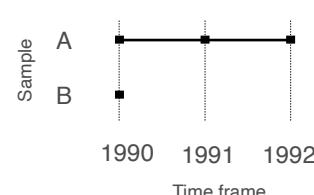
Note, if a data set is used that has multiple observations on the id-time unit, the function automatically aggregates the data set using the mean of condition 1 (**cond1**) and condition 2 (**cond2**).

If you store your results in an object, you can use **overview_print** to export them to a LaTeX output.

Generate Plots

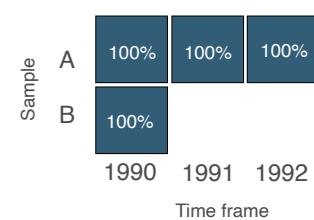
Sample overview

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

Number of NAs (% or total)

relative distribution

```
overview_na(toydata_with_na)
```

FALSE gives total number

```
overview_na(toydata_with_na,  
            perc = FALSE)
```

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

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 Development: : CHEAT SHEET



Package Structure

A package is a convention for organizing files into directories.

This sheet shows how to work with the 7 most common parts of an R package:



The contents of a package can be stored on disk as a:

- **source** - a directory with sub-directories (as above)
- **bundle** - a single compressed file (*.tar.gz*)
- **binary** - a single compressed file optimized for a specific OS

Or installed into an R library (loaded into memory during an R session) or archived online in a repository. Use the functions below to move between these states.



devtools:use_build_ignore("file")

Adds file to .Rbuildignore, a list of files that will not be included when package is built.

Setup (DESCRIPTION)

The **DESCRIPTION** file describes your work, sets up how your package will work with other packages, and applies a copyright.

- You must have a **DESCRIPTION** file
- Add the packages that yours relies on with `devtools::use_package()`
Adds a package to the Imports or Suggests field

CC0

No strings attached.

MIT

MIT license applies to your code if re-shared.

GPL-2

GPL-2 license applies to your code, *and all code anyone bundles with it*, if re-shared.

Package: mypackage

Title: Title of Package

Version: 0.1.0

Authors@R: person("Hadley", "Wickham", email = "hadley@me.com", role = c("aut", "cre"))

Description: What the package does (one paragraph)

Depends: R (>= 3.1.0)

License: GPL-2

LazyData: true

Imports:

dplyr (>= 0.4.0),

ggvis (>= 0.2)

Suggests:

knitr (>= 0.1.0)

Import packages that your package *must* have to work. R will install them when it installs your package.

Suggest packages that are not very essential to yours. Users can install them manually, or not, as they like.

Write Code (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 `devtools::create("path/to/name")`
Create a template to develop into a package.
- Save your code in **R/** as scripts (extension *.R*)

WORKFLOW

1. Edit your code.
2. Load your code with one of
`devtools::load_all()`
Re-loads all saved files in **R/** into memory.
Ctrl/Cmd + Shift + L (keyboard shortcut)
Saves all open files then calls `load_all()`.
3. Experiment in the console.
4. Repeat.
 - Use consistent style with [r-pkgs.had.co.nz/r.html#style](#)
 - Click on a function and press **F2** to open its definition
 - Search for a function with **Ctrl +**.



Visit [r-pkgs.had.co.nz](#) to learn much more about writing and publishing packages for R

Test (tests/)

Use **tests/** to store tests that will alert you if your code breaks.

- Add a **tests/** directory
- Import **testthat** with `devtools::use_testthat()`, which sets up package to use automated tests with **testthat**
- Write tests with **context()**, **test()**, and **expect** statements
- Save your tests as *.R* files in **tests/testthat/**

WORKFLOW

1. Modify your code or tests.
2. Test your code with one of
`devtools::test()`
Runs all tests in **tests/**
Ctrl/Cmd + Shift + T (keyboard shortcut)
3. Repeat until all tests pass

Example Test

```
context("Arithmetic")
test_that("Math works", {
  expect_equal(1 + 1, 2)
  expect_equal(1 + 2, 3)
  expect_equal(1 + 3, 4)
})
```

Expect statement	Tests
<code>expect_equal()</code>	is equal within small numerical tolerance?
<code>expect_identical()</code>	is exactly equal?
<code>expect_match()</code>	matches specified string or regular
<code>expect_output()</code>	prints specified output?
<code>expect_message()</code>	displays specified message?
<code>expect_warning()</code>	displays specified warning?
<code>expect_error()</code>	throws specified error?
<code>expect_is()</code>	output inherits from certain class?
<code>expect_false()</code>	returns FALSE?
<code>expect_true()</code>	returns TRUE?



Document (📄 man/)

📄 man/ contains the documentation for your functions, the help pages in your package.

- Use roxygen comments to document each function beside its definition
- Document the name of each exported data set
- Include helpful examples for each function

WORKFLOW

1. Add roxygen comments in your .R files
2. Convert roxygen comments into documentation with one of:

`devtools::document()`

Converts roxygen comments to .Rd files and places them in 📄 man/. Builds NAMESPACE.

Ctrl/Cmd + Shift + D (Keyboard Shortcut)

3. Open help pages with ? to preview documentation
4. Repeat

.Rd FORMATTING TAGS

\emph{italic text}	\email{name@@foo.com}
\strong{bold text}	\href{url}{display}
\code{function(args)}	\url{url}
\pkg{package}	
\dontrun{code}	\link[=dest]{display}
\dontshow{code}	\linkS4class{class}
\donttest{code}	\code{\link{function}}
\deqn{a + b (block)}	\code{\link[package]{function}}
\eqn{a + b (inline)}	\tabular{lcr}{ left \tab centered \tab right \cr cell \tab cell \tab cell \cr}

ROXYGEN2

The **roxygen2** package lets you write documentation inline in your .R files with a shorthand syntax. devtools implements roxygen2 to make documentation.



- Add roxygen documentation as comment lines that begin with #’.
- Place comment lines directly above the code that defines the object documented.
- Place a roxygen @ tag (right) after #’ to supply a specific section of documentation.
- Untagged lines 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.
#' @return The sum of \code{x} and \code{y}.
#' @examples
#' add(1, 1)
#' @export
add <- function(x, y) {
  x + y
}
```

COMMON ROXYGEN TAGS

@aliases	@inheritParams	@seealso	
@concepts	@keywords	@format	
@describeln	@param	@source	data
@examples	@rdname	@include	
@export	@return	@slot	S4
@family	@section	@field	RC

Teach (📄 vignettes/)

📄 vignettes/ holds documents that teach your users how to solve real problems with your tools.

- Create a 📄 vignettes/ directory and a template vignette with `devtools::use_vignette()`
Adds template vignette as vignettes/my-vignette.Rmd.
- Append YAML headers to your vignettes (like right)
- Write the body of your vignettes in R Markdown (rmarkdown.rstudio.com)

```
---
```

```
title: "Vignette Title"
author: "Vignette Author"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  \%VignetteIndexEntry{Vignette Title}
  \%VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
```

```
---
```

Add Data (📄 data/)

The 📄 data/ directory allows you to include data with your package.

- Save data as .Rdata files (suggested)
- Store data in one of **data/**, **R/Sysdata.rda**, **inst/extdata**
- Always use **LazyData: true** in your DESCRIPTION file.

`devtools::use_data()`

Adds a data object to data/ (R/Sysdata.rda if **internal = TRUE**)

`devtools::use_data_raw()`

Adds an R Script used to clean a data set to data-raw/. Includes data-raw/ on .Rbuildignore.

Store data in

- **data/** to make data available to package users
- **R/sysdata.rda** to keep data internal for use by your functions.
- **inst/extdata** to make raw data available for loading and parsing examples. Access this data with **system.file()**

Organize (📄 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
- Import objects from other packages with **package::object** (recommended) or **@import**, **@importFrom**, **@importClassesFrom**, **@importMethodsFrom** (not always recommended)

WORKFLOW

1. Modify your code or tests.
2. Document your package (`devtools::document()`)
3. Check NAMESPACE
4. Repeat until NAMESPACE is correct

SUBMIT YOUR PACKAGE

r-pkgs.had.co.nz/release.html



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

Score	Name	Short Description	Link
100	xmld	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	xmtr	Read, write and work with XML Data	16207
66	XML2R	Easier XML data collection	16204
65.7	xmldpcz	Implementation of the XML Procedure Call Protocol (XMLRPC)	16208
54.8	flatxml	Tools for working with XML Files as R DataFrames	4639

display = "viewer"

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:

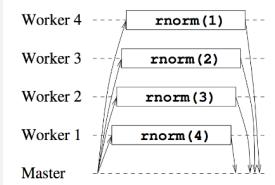
- sparklyr, iotools
- pbdr (programming with big data in R)

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(iertools)
foreach(s=isplitVector(1:10, chunks =2))%dopar% FUN
# e.g. FUN = sapply(s,"*",100)
future_sapply(..., future.scheduling = 1)
```

REST APIs with plumber: : CHEAT SHEET



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

HTTP Method: curl -v "http://httpbin.org/get"
#> GET / get HTTP/1.1
#> Host: httpbin.org
#> User-Agent: curl/7.55.1
#> Accept: */*

Request Body

Path: Path
HTTP Version: HTTP Version

Headers: Headers

Message body: Message body

RESPONSE

HTTP Version: #< HTTP/1.1 200 OK
Status code: Status code
Reason phrase: Reason phrase

Headers: Headers

Message body: Message body

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, "'"))
}
```

@decorators define API characteristics

HTTP Method: /<path> is used to define the location of the endpoint

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

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)
}
```

Filter name

Opt out of the log filter

Endpoint path

Serializer

Running Plumber APIs

Plumber APIs can be run programmatically from within an R session.

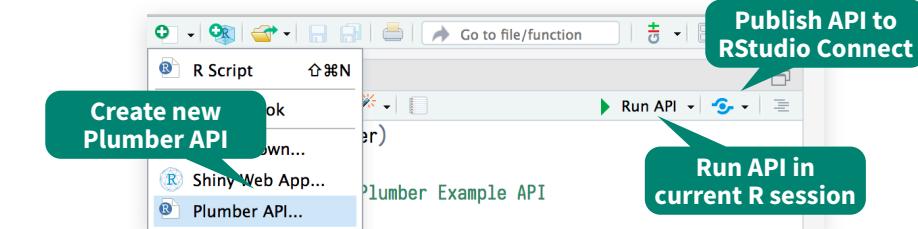
```
library(plumber)
plumb("plumber.R") %>%
  pr_run(port = 5762)
```

Path to API definition: Path to API definition

Specify API port: Specify API port

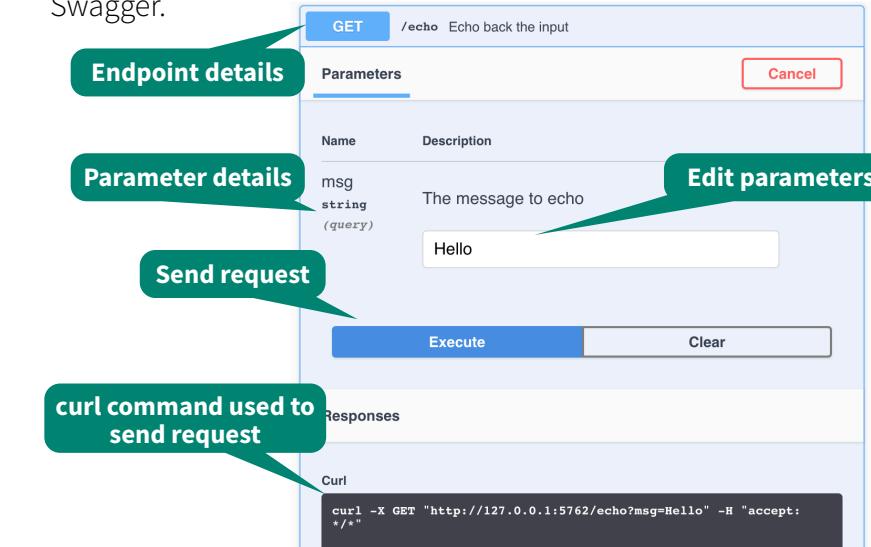
This runs the API on the host machine supported by the current R session.

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'\"]}"
```

Programmatic Plumber

Tidy Plumber

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_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()`.



Advanced Plumber

REQUEST and RESPONSE

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
<code>req</code>		
<code>req\$pr</code>	<code>plumber::pr()</code>	The Plumber router processing the request
<code>req\$body</code>	<code>list(a=1)</code>	Typically the same as <code>argsBody</code>
<code>req\$argsBody</code>	<code>list(a=1)</code>	The parsed body output
<code>req\$argsPath</code>	<code>list(c=3)</code>	The values of the path arguments
<code>req\$argsQuery</code>	<code>list(e=5)</code>	The parsed output from <code>req\$QUERY_STRING</code>
<code>req\$cookies</code>	<code>list(cook = "a")</code>	A list of cookies
<code>req\$REQUEST_METHOD</code>	"GET"	The method used for the HTTP request
<code>req\$PATH_INFO</code>	"/"	The path of the incoming HTTP request
<code>req\$HTTP_*</code>	"HTTP_USER_AGENT"	All of the HTTP headers sent with the request
<code>req\$bodyRaw</code>	<code>charToRaw("a=1")</code>	The <code>raw()</code> contents of the request body
<code>res</code>		
<code>res\$headers</code>	<code>list(header = "abc")</code>	HTTP headers to include in the response
<code>res\$setHeader()</code>	<code>setHeader("foo", "bar")</code>	Sets an HTTP header
<code>res\$setCookie()</code>	<code>setCookie("foo", "bar")</code>	Sets an HTTP cookie on the client
<code>res\$removeCookie</code>	<code>removeCookie("foo")</code>	Removes an HTTP cookie
<code>res\$body</code>	"{"a": [1]}"	Serialized output
<code>res\$status</code>	200	The response HTTP status code
<code>res\$toResponse()</code>	<code>toResponse()</code>	A list of status, headers, and body

ASYNC PLUMBER

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()
```

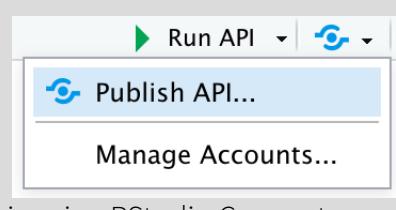
Identify the package name and API name

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.



Apply functions with purrr :: CHEAT SHEET



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)`

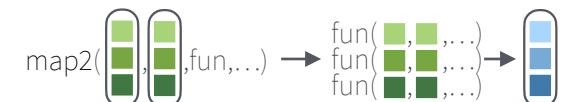
Function Shortcuts

Use `~.` with functions like **map()** that have single arguments.

`map(l, ~ . + 2)`
becomes
`map(l, function(x) x + 2)`

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)))`

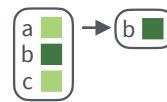
Use `~ .x .y` with functions like **imap()**. `.x` will get the list value and `.y` will get the index.

imap(list(a, b, c), ~ paste0(.y, ": ", .x))
outputs "index: value" for each item

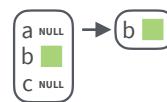


Work with Lists

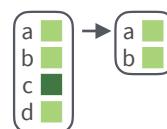
Filter



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)`



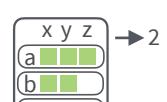
some(.x, .p, ...)
Do some elements pass a test?
`some(x, is.character)`



none(.x, .p, ...)
Do no elements pass a test?
`none(x, is.character)`

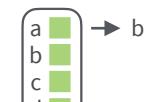


has_element(.x, .y)
Does a list contain an element?
`has_element(x, "foo")`

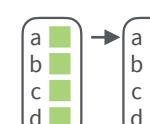


vec_depth(x)
Return depth (number of levels of indexes).
`vec_depth(x)`

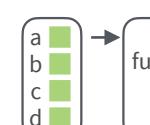
Index



pluck(.x, ..., .default=NULL)
Select an element by name or index. Also **attr_getter()** and **chuck()**.
`pluck(x, "b")`
`x %>% pluck("b")`

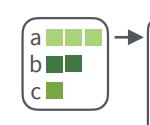


assign_in(x, where, value)
Assign a value to a location using pluck selection.
`assign_in(x, "b", 5)`
`x %>% assign_in("b", 5)`



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)`

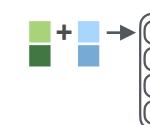
Reshape



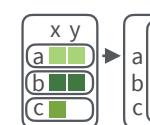
flatten(.x) Remove a level of indexes from a list.
Also **flatten_chr()** etc.
`flatten(x)`



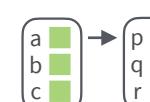
array_tree(array, margin = NULL) Turn array into list.
Also **array_branch()**.
`array_tree(x, margin = 3)`



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)`

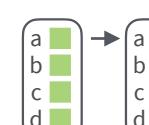


set_names(x, nm = x)
Set the names of a vector/list directly or with a function.
`set_names(x, c("p", "q", "r"))`
`set_names(x, tolower)`

Modify



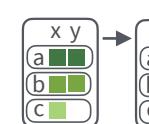
modify(.x, .f, ...) Apply a function to each element. Also **modify2()**, and **imodify()**.
`modify(x, ~.+ 2)`



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)`

Combine



append(x, values, after = length(x)) Add values to end of list.
`append(x, list(d = 1))`



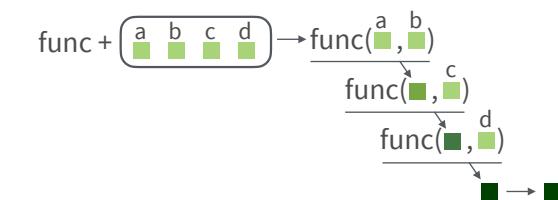
prepend(x, values, before = 1) Add values to start of list.
`prepend(x, list(d = 1))`



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)`



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.

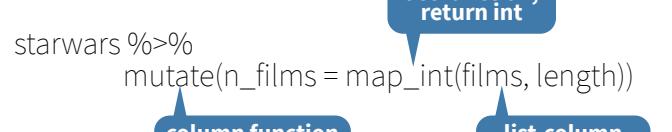
WORK WITH LIST-COLUMNS

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.

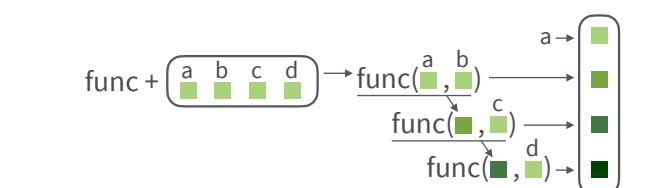
map(), **map2()**, or **pmap()** return lists and will **create new list-columns**.



Suffixed map functions like **map_int()** return an atomic data type and will **simplify list-columns into regular columns**.



accumulate(.x, .f, ..., .init) Reduce a list, but also return intermediate results. Also **accumulate2()**.
`accumulate(x, sum)`



quanteda Cheat Sheet

Quantitative Analysis of Textual Data

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
## 1 1789-Washington      1     1     2      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/remove/select/toupper/tolower 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

randomizr:: CHEAT SHEET

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)
```

Basic Regular Expressions in R

Cheat Sheet

Character Classes

<code>[:digit:]</code> or <code>\d</code>	Digits; [0-9]
<code>\D</code>	Non-digits; [^0-9]
<code>[:lower:]</code>	Lower-case letters; [a-z]
<code>[:upper:]</code>	Upper-case letters; [A-Z]
<code>[:alpha:]</code>	Alphabetic characters; [A-z]
<code>[:alnum:]</code>	Alphanumeric characters [A-z0-9]
<code>\w</code>	Word characters; [A-z0-9_]
<code>\W</code>	Non-word characters
<code>[:xdigit:]</code> or <code>\x</code>	Hexadec. digits; [0-9A-Fa-f]
<code>[:blank:]</code>	Space and tab
<code>[:space:]</code> or <code>\s</code>	Space, tab, vertical tab, newline, form feed, carriage return
<code>\S</code>	Not space; [^[:space:]]
<code>[:punct:]</code>	Punctuation characters; !#\$%&'()*+, -./; <=>?@[]^_`{ }~
<code>[:graph:]</code>	Graphical characters; [[:alnum:][:punct:]]
<code>[:print:]</code>	Printable characters; [[:alnum:][:punct:]\s]
<code>[:cntrl:]</code> or <code>\c</code>	Control characters; \n, \r etc.

Special Metacharacters

<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\v</code>	Vertical tab
<code>\f</code>	Form feed

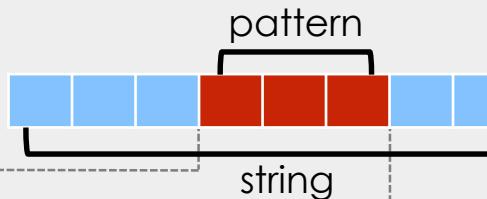
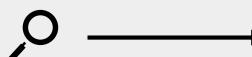
Lookarounds and Conditionals*

<code>(?=)</code>	Lookahead (requires PERL = TRUE), e.g. (?=yx): position followed by 'xy'
<code>(?!)</code>	Negative lookahead (PERL = TRUE); position NOT followed by pattern
<code>(?<=)</code>	Lookbehind (PERL = TRUE), e.g. (?<=yx): position following 'xy'
<code>(?<!)</code>	Negative lookbehind (PERL = TRUE); position NOT following pattern
<code>?(if)then</code>	If-then-condition (PERL = TRUE); use lookaheads, optional char. etc in if-clause
<code>?(if)then else</code>	If-then-else-condition (PERL = TRUE)

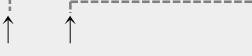
*see, e.g. <http://www.regular-expressions.info/lookaround.html>
<http://www.regular-expressions.info/conditional.html>

Functions for Pattern Matching

Detect pattern



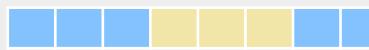
Locate pattern



Extract pattern



Replace pattern



```
> string <- c("Hipopopotamus", "Rhymenoceros", "time for bottomless lyrics")
> pattern <- "t.m"
```

Detect Patterns

`grep(pattern, string)`

[1] 1 3

`grep(pattern, string, value = TRUE)`

[1] "Hipopopotamus"
[2] "time for bottomless lyrics"

`grepl(pattern, string)`

[1] TRUE FALSE TRUE

`stringr::str_detect(string, pattern)`

[1] TRUE FALSE TRUE

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

Split a String using a Pattern

`strsplit(string, pattern)` or `stringr::str_split(string, pattern)`

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

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

\E End of a word

Quantifiers

* Matches at least 0 times

+ Matches at least 1 time

? Matches at most 1 time; optional string

{n} Matches exactly n times

{n,m} Matches at least n times

{n,m} Matches between n and m times

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`.

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`.

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`.



Use Python with R with reticulate :: CHEAT SHEET

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.

```

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 ````{python, echo = FALSE}
8 import seaborn as sns
9 fmri = sns.load_dataset("fmri")
10 ````

11 ````{r}
12 f1 <- subset(py$fmri, region == "parietal")
13 ````

14 ````{python}
15 import matplotlib as mpl
16 sns.lmplot("timepoint","signal", data=r.f1)
17 mpl.pyplot.show()
18 ````

19

```

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))`

```

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

```

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$x`



Python 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.

Python REPL

A REPL (Read, Eval, Print Loop) is a command line where you can run Python code and view the results.

1. Open in the console with **repl_python()**, or by running code in a Python script with **Cmd + Enter** (**Ctrl + Enter**).
2. Type commands at **>>>** prompt.
3. Press **Enter** to run code.
4. 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

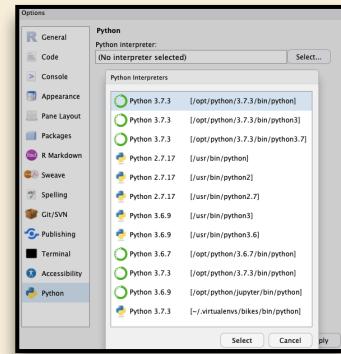
- **install_python(version, list = FALSE, force = FALSE)** Download and install Python.
install_python("3.6.13")
- **py_available(initialize = FALSE)** Check if Python is available on your system. Also **py_module_available()** and **py_numpy_module()**. py_available()
- **py_discover_config()** Return 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()

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()**.

1. 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")

2. 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")

3. Within virtualenvs and conda envs that carry the same name as the imported module. e.g. ~/anaconda/envs/nltk for **import("nltk")**

4. At the location of the Python binary discovered on the system PATH (i.e. **Sys.which("python")**)

5. At customary locations for Python, e.g. /usr/local/bin/python, /opt/local/bin/python...

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")

rmarkdown :: CHEAT SHEET

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
```
```

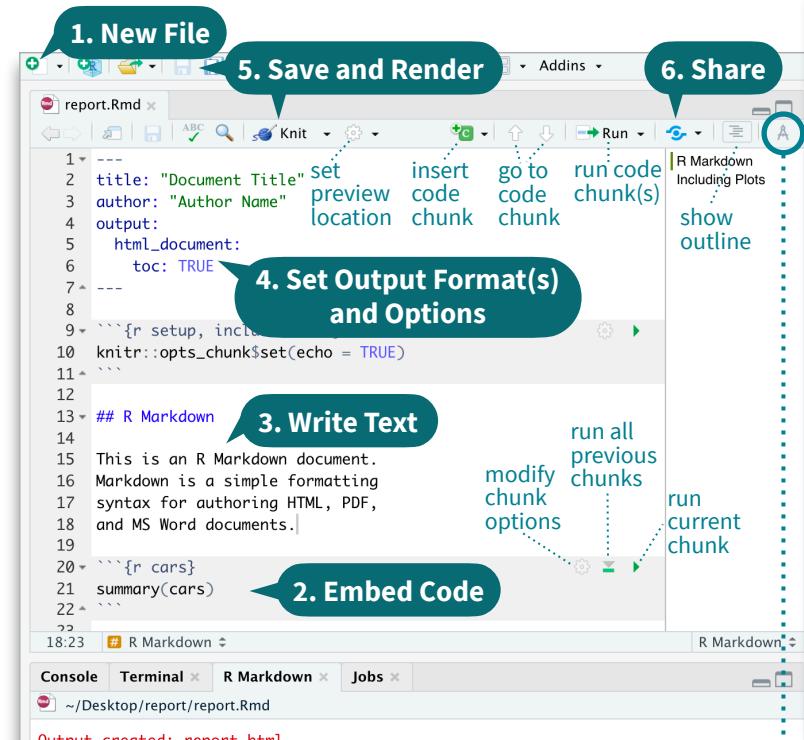
INLINE CODE

Insert `{{r <code>}}` into text sections. Code is evaluated at render and results appear as text.

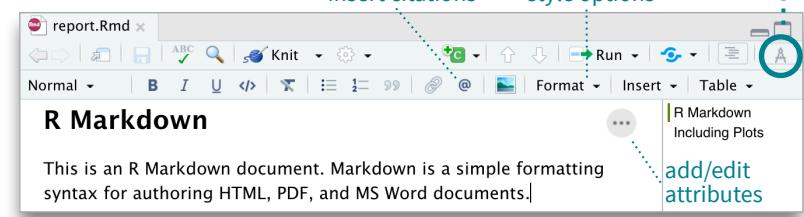
"Built with `{{r getRversion()}}`" --> "Built with 4.1.0"



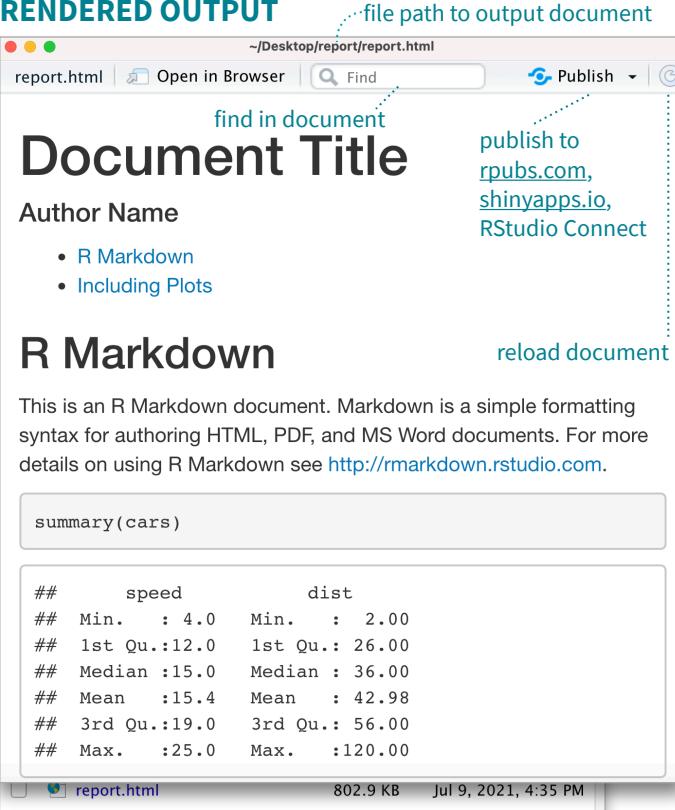
SOURCE EDITOR



VISUAL EDITOR



RENDERED OUTPUT



Write with Markdown

The syntax on the left renders as the output on the right.

Plain text.

Plain text.

End a line with two spaces to start a new paragraph.

End a line with two spaces to start a new paragraph.

Also end with a backslash\ to make a new line.

Also end with a backslash\ to make a new line.

italics* and ****bold****

italics and **bold**

superscript²/subscript₂

superscript²/subscript₂

~~strikethrough~~

strikethrough

escaped: `*` \`

escaped: * \`

endash: --, emdash: ---

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.]

[This is another link][id].

This is another link.

<http://www.rstudio.com/>

This is a link.

This is another link.



Caption.

verbatim code

multiple lines of verbatim code

> block quotes

block quotes

multiple lines of verbatim code

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 |

| Right | Left | Default | Center |
|-------|------|---------|--------|
| 12 | 12 | 12 | 12 |
| 123 | 123 | 123 | 123 |
| 1 | 1 | 1 | 1 |

HTML Tabsets

```
# Results {.tabset}
## Plots text
text
```

Tables
more text

Results

| | |
|-------|--------|
| Plots | Tables |
|-------|--------|

text

| 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())`

Insert Tables

Output data frames as tables using `kable(data, caption)`.

```
```{r}
data <- faithful[1:4,]
knitr::kable(data,
 caption = "Table with kable")
```
```

| Table with kable | |
|------------------|---------|
| eruptions | waiting |
| 3.600 | 79 |
| 1.800 | 54 |
| 3.333 | 74 |
| 2.283 | 62 |

Other table packages include `flextable`, `gt`, and `kableExtra`.





Set Output Formats and their Options in YAML

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`.

| IMPORTANT OPTIONS | DESCRIPTION | HTML | PDF | MS Word | MS PPT |
|---------------------|--|---------|-----|---------|--------|
| 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`

More Header Options

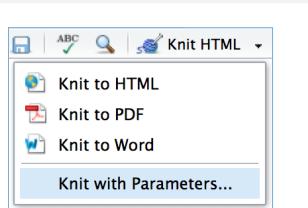
PARAMETERS

Parameterize your documents to reuse with new inputs (e.g., data, values, etc.).

1. **Add parameters** in the header as sub-values of `params`.
2. **Call parameters** in code using `params$<name>`.
3. **Set parameters** with Knit with Parameters or the `params` argument of `render()`.

REUSABLE TEMPLATES

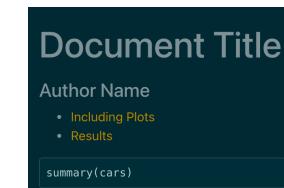
1. **Create a new package** with a `inst/rmarkdown/templates` directory.
2. **Add a folder** containing `template.yaml` (below) and `skeleton.Rmd` (template contents).
3. **Install** the package to access template by going to **File > New R Markdown > From Template**.



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/.

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.

.my-css-tag ...
This is a div with some text in it.

Render

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/



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)
})
```

speed	dist
1	4.00
2	4.00
3	7.00
4	7.00
5	8.00
	2.00
	10.00
	4.00
	22.00
	16.00

Also see Shiny Prerendered for better performance.

[rstudio.com/rmarkdown/rstudiocom/authoring\\_shiny\\_prerendered](https://rstudio.com/rmarkdown/rstudiocom-authoring_shiny_prerendered)

Embed a complete app into your document with `shiny::shinyAppDir()`. More at [bookdown.org/yihui/rmarkdown/shiny-embedded.html](https://bookdown.org/yihui/rmarkdown/shiny-embedded.html).

# rmarkdown :: CHEAT SHEET

## 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
```
```

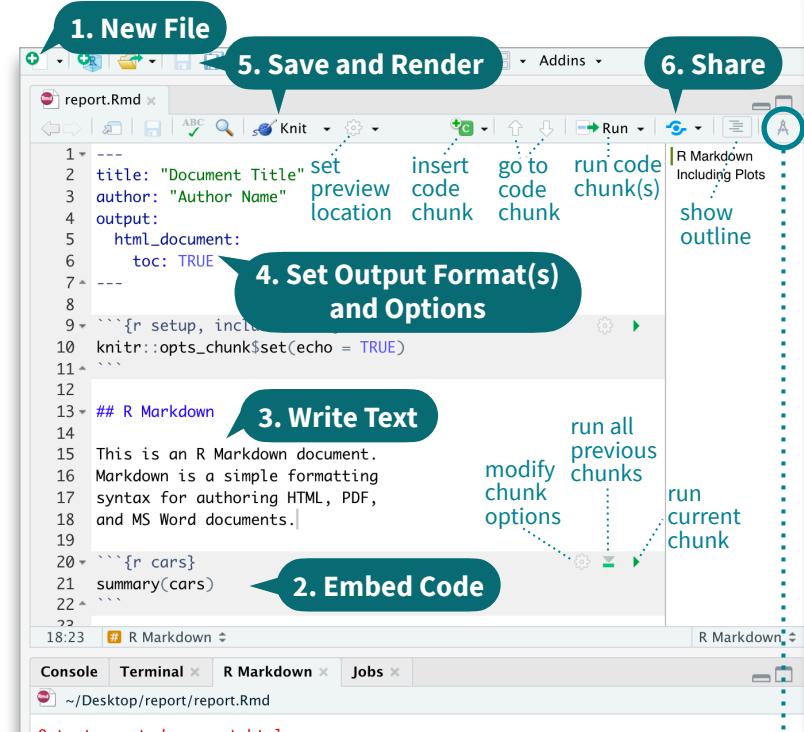
### INLINE CODE

Insert `r <code>` into text sections. Code is evaluated at render and results appear as text.

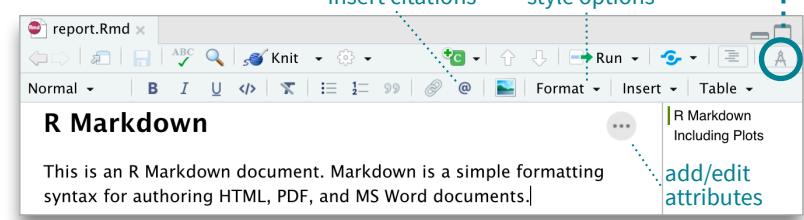
"Built with `r getRversion()`" --> "Built with 4.1.0"



### SOURCE EDITOR



### VISUAL EDITOR



### RENDERED OUTPUT



## Write with Markdown

The syntax on the left renders as the output on the right.

Plain text.

Plain text.

End a line with two spaces to start a new paragraph.

End a line with two spaces to start a new paragraph.

Also end with a backslash\ to make a new line.

Also end with a backslash\ to make a new line.

**italics\*** and **\*\*bold\*\***

**italics** and **bold**

superscript<sup>2</sup>/subscript<sub>2</sub>

superscript<sup>2</sup>/subscript<sub>2</sub>

~~strikethrough~~

strikethrough

escaped: `\*` \`

escaped: \* \`

endash: --, emdash: ---

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].

This is another link.

<http://www.rstudio.com/>

This is a link.

This is another link.



Caption.

verbatim code

multiple lines of verbatim code

> block quotes

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 Tabsets

```
Results {.tabset}
Plots text
text
```

## Tables
more text

### Results

|       |        |
|-------|--------|
| Plots | Tables |
|-------|--------|

text





# Set Output Formats and their Options in YAML

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`.

| IMPORTANT OPTIONS   | DESCRIPTION                                                                            | HTML    | PDF | MS Word | MS PPT |
|---------------------|----------------------------------------------------------------------------------------|---------|-----|---------|--------|
| 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`

## More Header Options

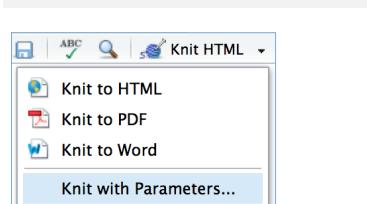
### PARAMETERS

Parameterize your documents to reuse with new inputs (e.g., data, values, etc.).

1. **Add parameters** in the header as sub-values of `params`.
2. **Call parameters** in code using `params$<name>`.
3. **Set parameters** with Knit with Parameters or the `params` argument of `render()`.

### 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).
3. **Install** the package to access template by going to **File > New R Markdown > From Template**.



### 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.

.my-css-tag ...  
This is a div with some text in it.

## Render

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)
})
```

speed	dist
1	4.00
2	4.00
3	7.00
4	7.00
5	8.00
	2.00
	10.00
	4.00
	22.00
	16.00

Also see Shiny Prerendered for better performance.

rmarkdown.rstudio.com/authoring_shiny_prerendered

Embed a complete app into your document with `shiny::shinyAppDir()`. More at bookdown.org/yihui/rmarkdown/shiny-embedded.html.

Add silhouettes with rphylopic :: CHEAT SHEET

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 uuid code based on common name or taxonomy of a species. The options can be namebankID, type, names, root, uri.

- **name_get**(uuid, options)

Get information on a name using the uuid code. The options can be citationStart, html, namebankID, root.

- **name_images**(uuid, options = 'credit')

Searches for different images for a taxonomic name.

- **name_taxonomy**(uuid, options, as)

Returns taxonomic name based on uuid code. Options can be string, and as can be list, table, json.

- **name_taxonomy_many**(uuid, options, as)

Returns taxonomic names for two or more concatenated (c()) uuid codes.

- **name_taxonomy_sources**(uuid)

Gives information on the sources for a name's taxonomy given a uuid.

2. Work with uBio data

- **ubio_get**(namebankID)

Retrieve the uuid code based on the namebankID number.

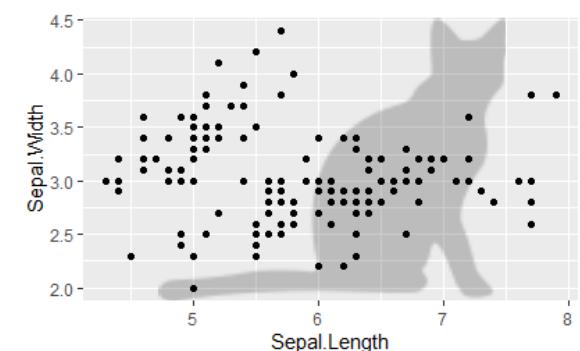
Plot silhouettes

1. Plot a silhouette behind a plot

- **ggplot**

```
library(ggplot2)  
cat <- image_data("23cd6aa4-9587-4a2e-  
8e26-de42885004c9", size = 128)[[1]]
```

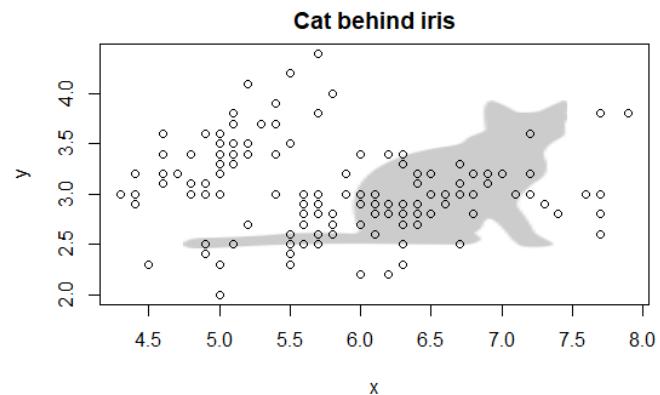
```
ggplot(data = iris,  
       aes(x = Sepal.Length,  
            y = Sepal.Width))+  
  geom_point() +  
  add_phylopic(cat, alpha = 0.2)
```



- **Base plot**

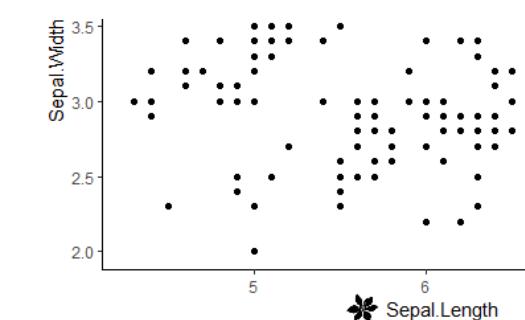
```
cat <- image_data("23cd6aa4-9587-4a2e-  
8e26-de42885004c9", size = 128)[[1]]
```

```
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

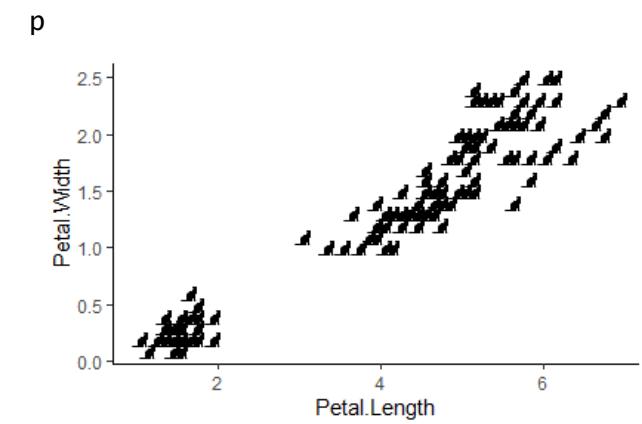
```
ggpubr::ggarrange(plot) +  
  add_phylopic(irisimg,  
              alpha = 1,  
              x = 0.43,  
              y = 0.05,  
              ysize = 0.06)
```



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)
```



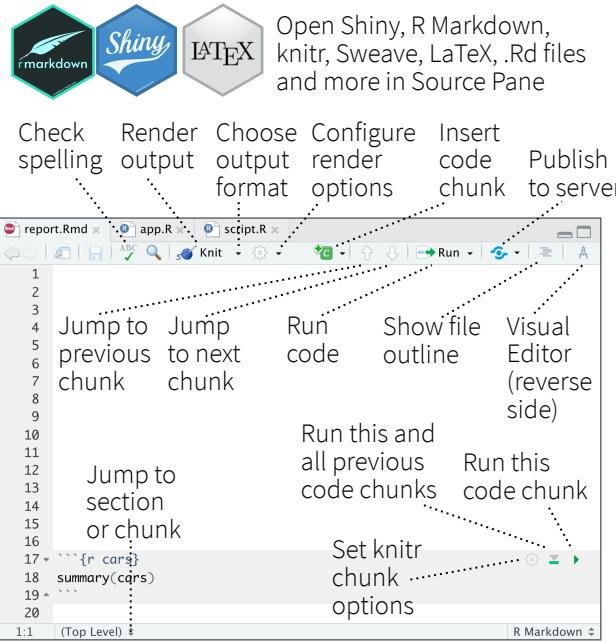
Citation

Don't forget to cite rphylopic. See how here:

```
citation("rphylopic")
```

RStudio IDE :: CHEAT SHEET

Documents and Apps



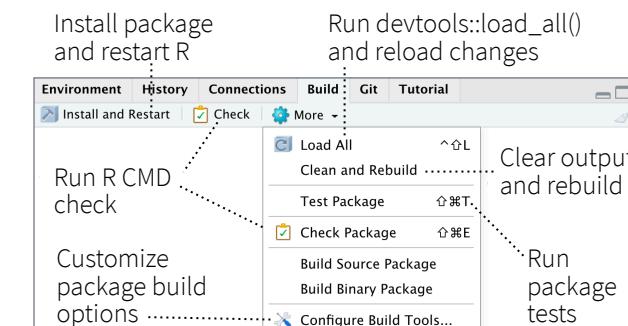
Package Development

Create a new package with **File > New Project > New Directory > R Package**

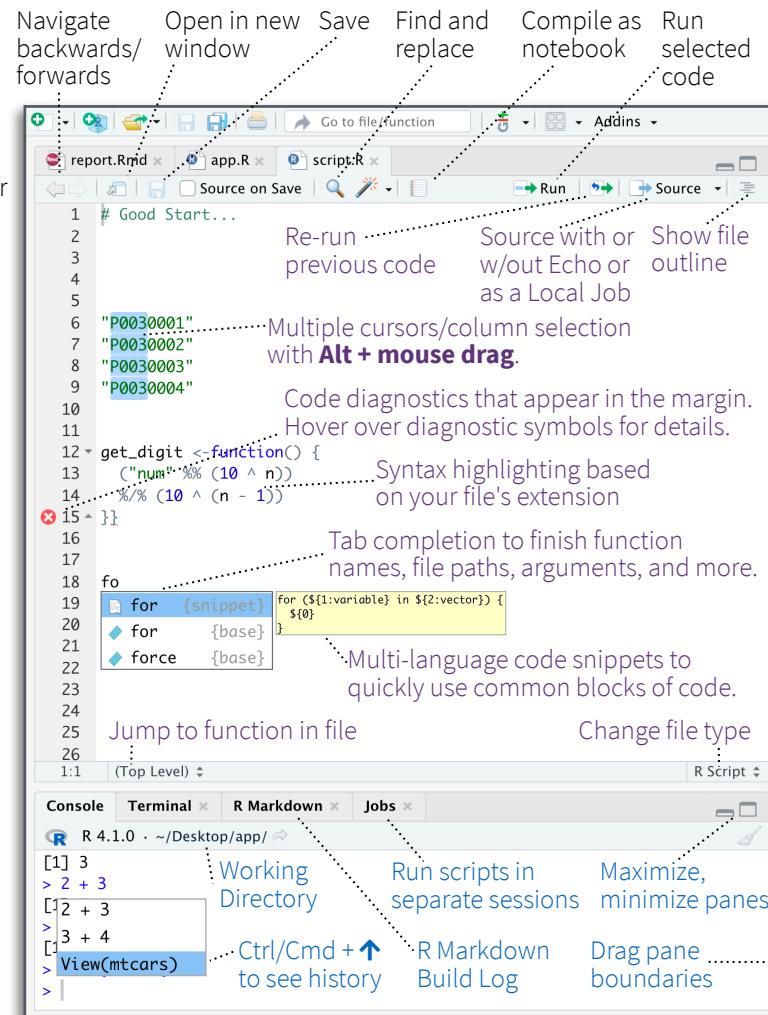
Enable roxygen documentation with **Tools > Project Options > Build Tools**

Roxygen guide at **Help > Roxygen Quick Reference**

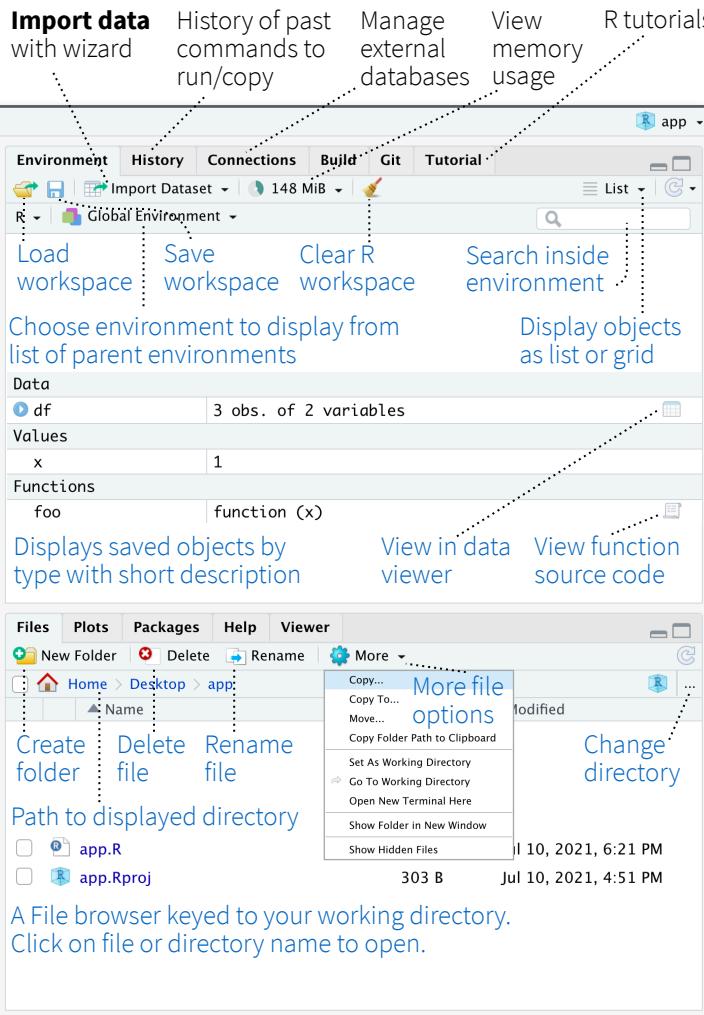
See package information in the **Build Tab**



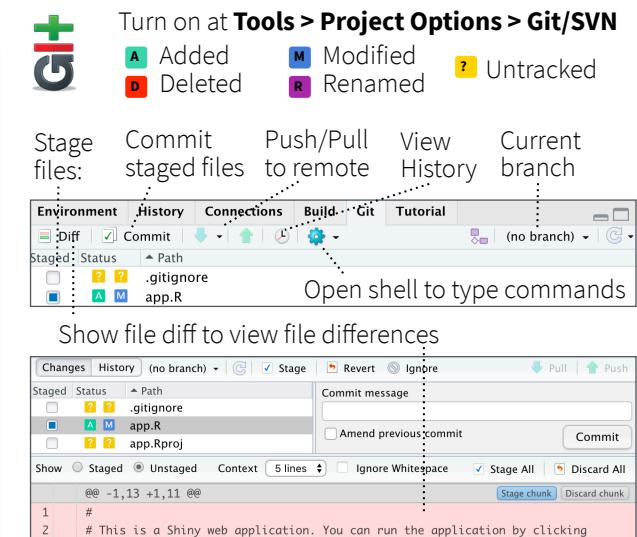
Source Editor



Tab Panes

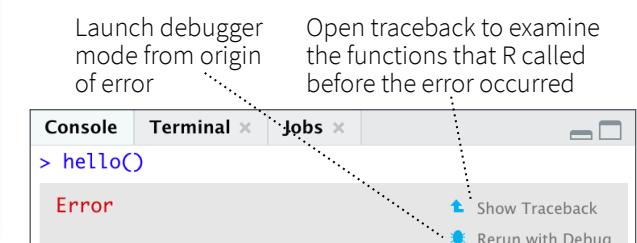


Version Control



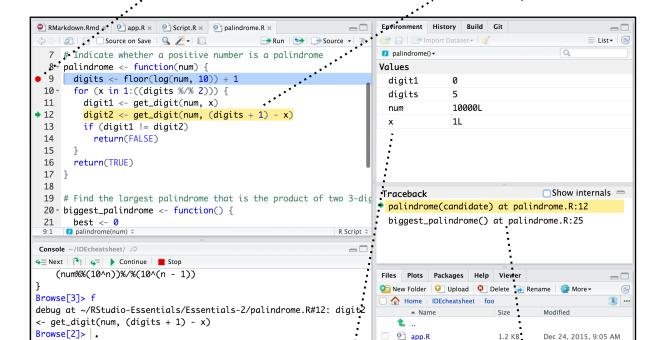
Debug Mode

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.

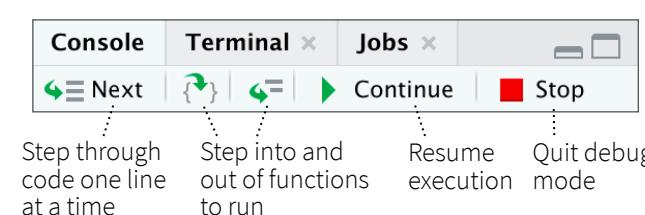
Highlighted line shows where execution has paused



Run commands in environment where execution has paused

Examine variables in executing environment

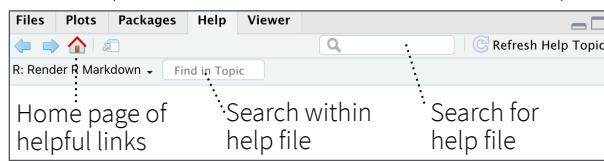
Select function in traceback to debug



RStudio opens plots in a dedicated **Plots** pane



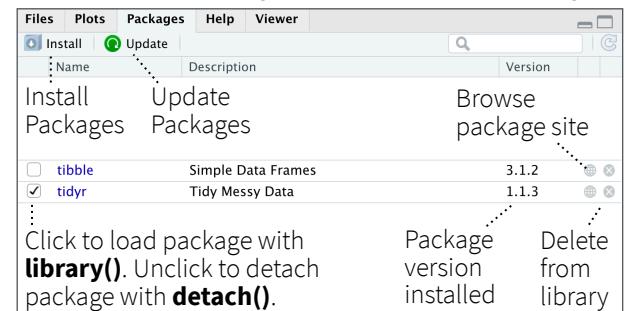
RStudio opens documentation in a dedicated **Help** pane



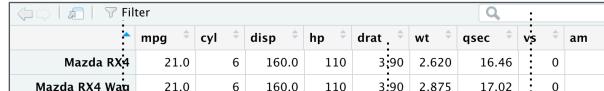
Viewer pane displays HTML content, such as Shiny apps, RMarkdown reports, and interactive visualizations



GUI **Package** manager lists every installed package



View(<data>) opens spreadsheet like view of data set





Keyboard Shortcuts

RUN CODE

Search command history
Interrupt current command
Clear console

Windows/Linux	Mac
Ctrl+↑	Cmd+↑
Esc	Esc
Ctrl+L	Ctrl+L

Navigate Code

Go to File/Function

Windows/Linux	Mac
Ctrl+. .	Ctrl+. .

Write Code

Attempt completion

Tab or Ctrl+Space	Tab or Ctrl+Space
Insert <- (assignment operator)	Alt+-
Insert %>% (pipe operator)	Ctrl+Shift+M
(Un)Comment selection	Ctrl+Shift+C

Windows/Linux	Mac
Ctrl+Shift+L	Cmd+Shift+L
Ctrl+Shift+T	Cmd+Shift+T
Ctrl+Shift+D	Cmd+Shift+D

MAKE PACKAGES

Load All (devtools)
Test Package (Desktop)
Document Package

DOCUMENTS AND APPS

Knit Document (knitr)	Ctrl+Shift+K	Cmd+Shift+K
Insert chunk (Sweave & Knitr)	Ctrl+Alt+I	Cmd+Option+I
Run from start to current line	Ctrl+Alt+B	Cmd+Option+B

MORE KEYBOARD SHORTCUTS

Keyboard Shortcuts Help	Alt+Shift+K	Option+Shift+K
Show Command Palette	Ctrl+Shift+P	Cmd+Shift+P

View the Keyboard Shortcut Quick Reference with **Tools > Keyboard Shortcuts** or **Alt/Option + Shift + K**



Search for keyboard shortcuts with **Tools > Show Command Palette** or **Ctrl/Cmd + Shift + P**.

RStudio Workbench

WHY RSTUDIO WORKBENCH?

Extend the open source server with a commercial license, support, and more:

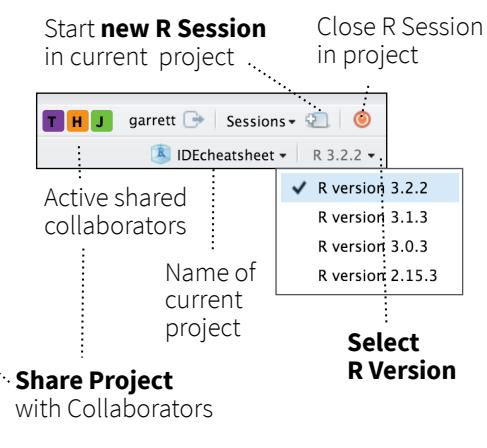
- open and run multiple R sessions at once
- tune your resources to improve performance
- administrative tools for managing user sessions
- collaborate real-time with others in shared projects
- switch easily from one version of R to a different version
- integrate with your authentication, authorization, 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.



Visual Editor

The screenshot shows the RStudio Visual Editor interface. A R Markdown document titled "report.Rmd" is open. Various keyboard shortcuts are overlaid on the interface, including:

- 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)
- File outline
- Lists and block quotes
- Links
- Citations
- Images
- More formatting
- Insert blocks, citations, equations, and special characters
- Insert and edit tables
- Insert verbatim code
- Clear formatting
- Add/Edit attributes
- Set knitr chunk options
- Run this and all previous code chunks
- Run this code chunk
- Jump to chunk or header



Run Remote Jobs

Run R on remote clusters (Kubernetes/Slurm) via the Job Launcher

The screenshot shows the RStudio Job Launcher interface. It displays a list of jobs:

- fast.R (Running, Local, 0:09)
- sleepy.R (Succeeded 11:22 AM, Local, 0:41)
- sleepy.R (Idle, KubernetesX, Waiting)

Options available in the interface include:

- Run
- Source
- Source with Echo
- Source as Launcher Job...
- Source as Local Job...
- Start Launcher Job
- Sorted by submission time
- Monitor launcher jobs
- Launch a job
- Run launcher jobs remotely

SAS <-> R :: CHEAT SHEET

Introduction

This guide aims to familiarise SAS users with R.
R examples make use of tidyverse collection of packages.

Install tidyverse: `install.packages("tidyverse")`
Attach tidyverse packages for use: `library(tidyverse)`

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 =
"Moderate";
else weather = "Cold";
run;
```

```
new_data <- old_data %>%
mutate(weather = case_when(
temp > 20 ~ "Warm",
temp > 10 ~ "Moderate",
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

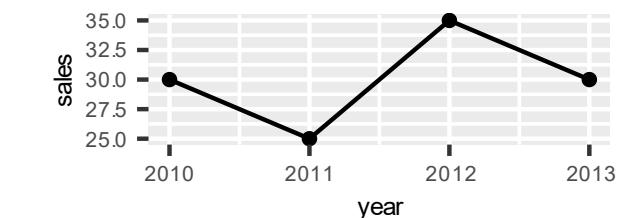
```
data new_data ;
merge data_1 (in= in_1) data_2 ;
by id ;
if in_1 ;
run;
```

```
new_data <- left_join( data_1 , data_2 , by = "id" )
```

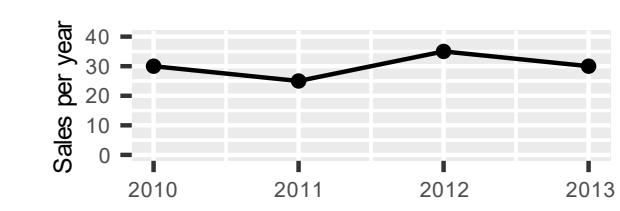
C.f. full_join(), right_join(), inner_join()

Some plotting in R

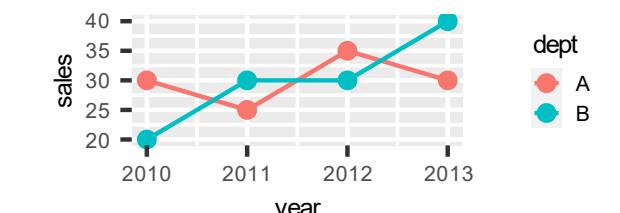
```
ggplot( my_data , aes( year , sales ) ) +
geom_point() + geom_line()
```



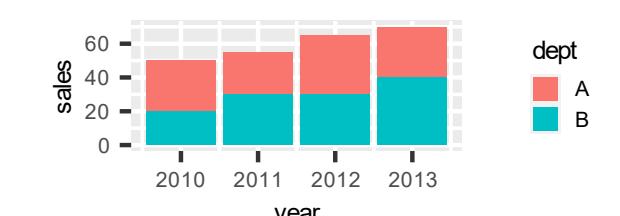
```
ggplot( my_data , aes( year , sales ) ) +
geom_point() + geom_line() + ylim(0, 40) +
labs(x = "" , y = "Sales per year")
```



```
ggplot(my_data, aes( year, sales, colour = dept ) ) +
geom_point() + geom_line()
```

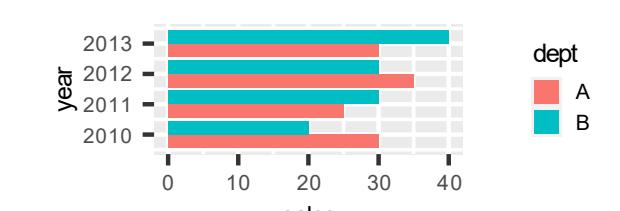


```
ggplot( my_data , aes( year, sales, fill = dept ) ) +
geom_col()
```



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

Sorting and Row-Wise Operations

```

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

```

Dealing with strings

```

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;

new_data <- old_data %>%
  filter(str_detect(job_title , "Health" ))

new_data <- old_data %>%
  filter(str_detect(job_title , "^Health"))

Use ^ for start of string, $ for end of string, e.g. "Health$"

new_data <- old_data %>%
  mutate(substring = str_sub( big_string , 3 , 6 ))

Returns characters 3 to 6. Note SAS uses <start>, <length>, R uses <start>, <end>

new_data <- old_data %>%
  mutate(address = str_replace_all( address , "Street" , "St" ))

C.f. str_replace( ) for first instance of pattern only

new_data <- old_data %>%
  mutate(full_name = str_c( first_name , surname , sep = " " ))

Drop sep = " " for equivalent to cats( ) in SAS

new_data <- old_data %>%
  mutate(first_word = word( sentence , 1 ))

R example preserves punctuation at the end of words, SAS doesn't

new_data <- old_data %>%
  mutate(house_number = str_extract( address , "\d*" ))

Wide range of regexps in both languages, this example extracts digits only

```

File operations

Operate in 'Work' library.
Use **libname** to define file locations

Operate in a particular 'working directory' (identify using **getwd()**)
Move to other locations using **setwd()**

```

libname library_name "file_location";
data library_name.saved_data;
  set data_in_use;
run;

```

```

save(data_in_use , file="file_location/saved_data.rda")
or
setwd("file_location")
save( data_in_use , file = "saved_data.rda")

```

```

libname library_name "file_location";
data data_in_use ;
  set library_name.saved_data ;
run;

```

```

load("file_location/saved_data.rda" )
or
setwd("file_location")
load("saved_data.rda")

```

save() can store multiple data frames in a single .rda file, load() will restore all of these

```

proc import datafile = "my_file.csv"
  out = my_data dbms = csv;
run;

```

```

my_data <- read_csv("my_file.csv")

```

Both examples assume column headers in csv file

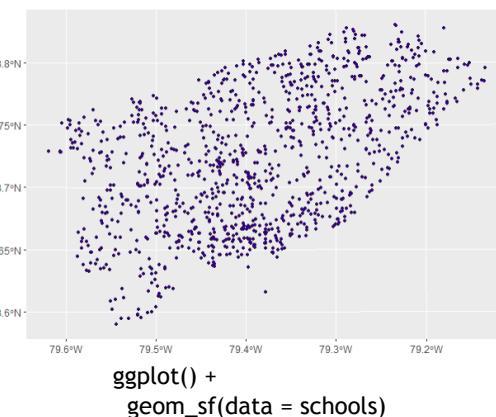
Spatial manipulation with sf: : CHEAT SHEET



The sf package provides a set of tools for working with geospatial vectors, i.e. points, lines, polygons, etc.

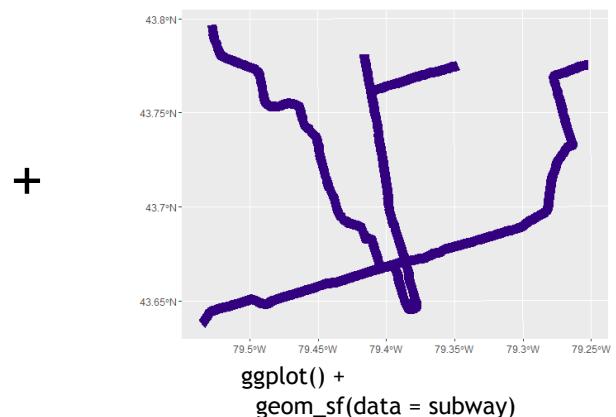
Geometric confirmation

- st_contains(x, y, ...) Identifies if x is within y (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



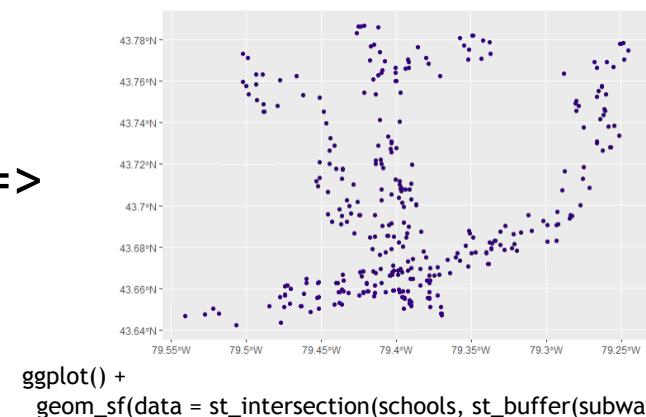
Geometric operations

- 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



Spatial manipulation with sf: : CHEAT SHEET



The sf package provides a set of tools for working with geospatial vectors, i.e. points, lines, polygons, etc.

Geometry operations

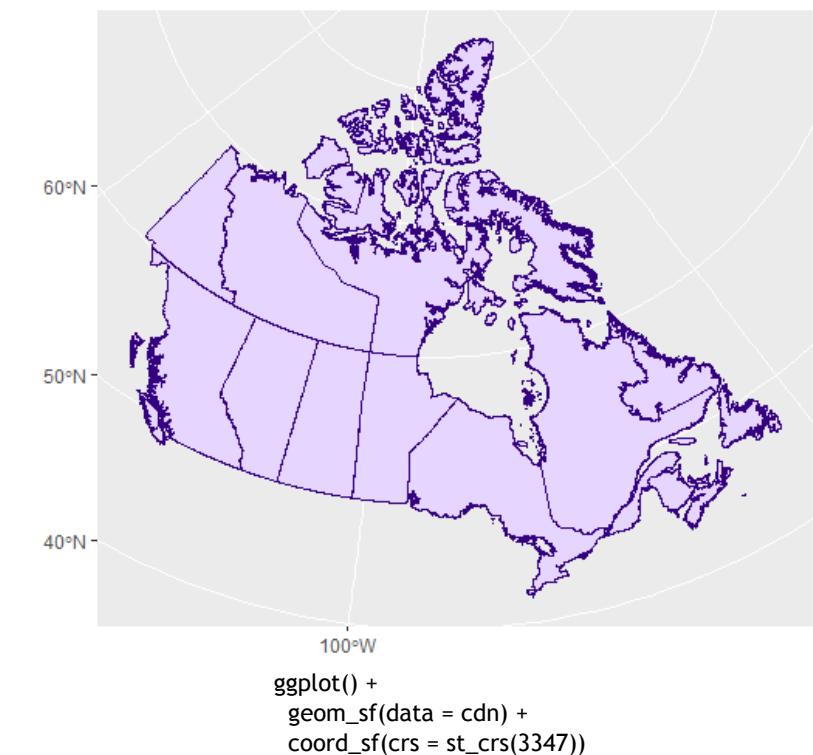
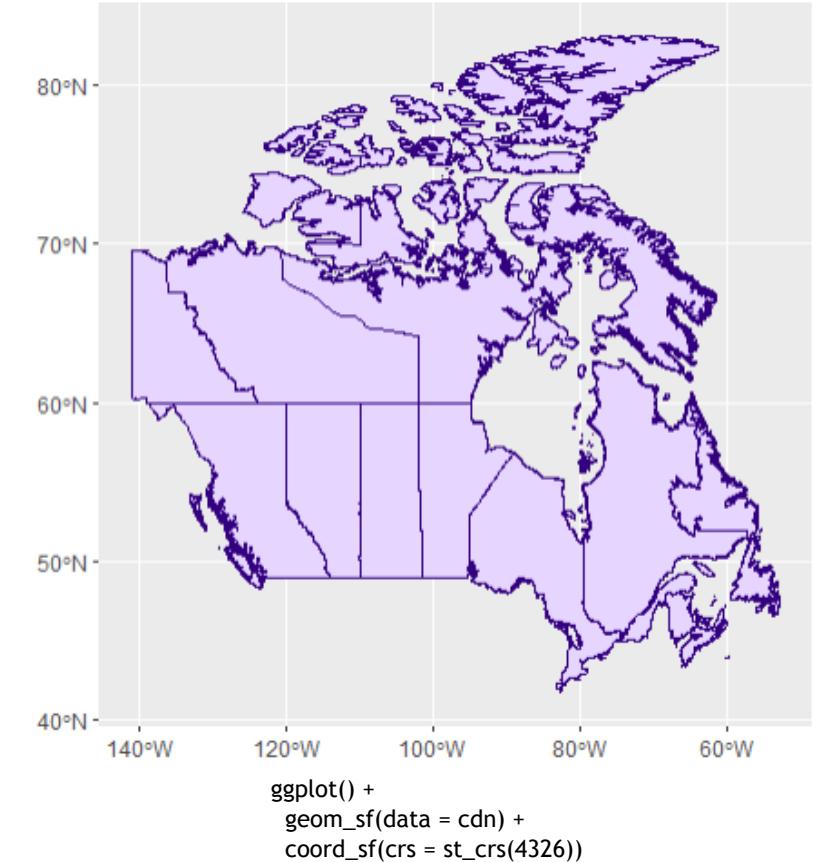
- `st_contains(x, y, ...)` Identifies if x is within y (i.e. point within polygon)
- `st_crop(x, y, ..., xmin, ymin, xmax, ymax)` Creates geometry of x that intersects a specified rectangle
- `st_difference(x, y)` Creates geometry from x that does not intersect with y
- `st_intersection(x, y)` Creates 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



Shiny :: CHEAT SHEET



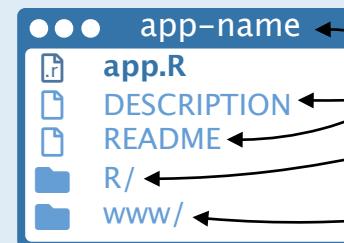
Building an App

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)
```

In **ui** nest R functions to build an HTML interface

Customize the UI with **Layout Functions**

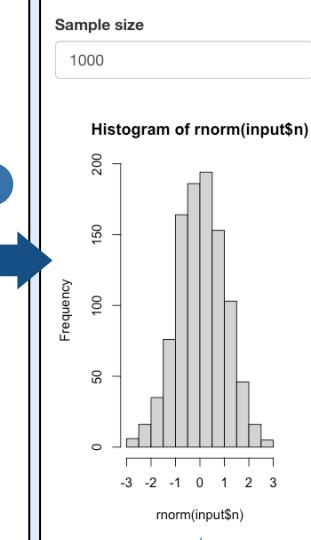
Add Inputs with ***Input()** functions

Add Outputs with ***Output()** functions

Tell the **server** how to render outputs and respond to inputs with R

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/

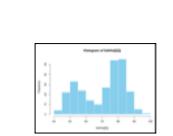
3. **Build your own Shiny Server** rstudio.com/products/shiny/shiny-server/

Outputs

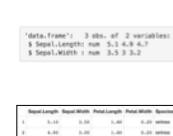
render*() and ***Output()** functions work together to add R output to the UI.



DT::renderDataTable(expr, options, searchDelay, callback, escape, env, quoted, outputArgs)



renderImage(expr, env, quoted, deleteFile, 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

actionLink(inputId, label, icon, ...)

✓ Choice 1

✓ Choice 2

□ Choice 3

✓ Check me

checkboxGroupInput(inputId, label, choices, selected, inline, width, choiceNames, choiceValues)

checkboxInput

checkboxInput(inputId, label, value, width)

dateInput

dateInput(inputId, label, value, min, max, format, startview, weekstart, language, width, autoclose, datesdisabled, daysofweekdisabled)

dateRangeInput

dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator, width, autoclose)

Choose File

fileInput(inputId, label, multiple, accept, width, buttonLabel, placeholder)

1

numericInput(inputId, label, value, min, max, step, width)

.....

passwordInput(inputId, label, value, width, placeholder)

Choice A

Choice B

Choice C

Choice 1

Choice 2

Choice 3

0 1 2 3 4 5 6 7 8 9 10

0 1 2 3 4 5 6 7 8 9 10

Apply Changes

Enter text

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)

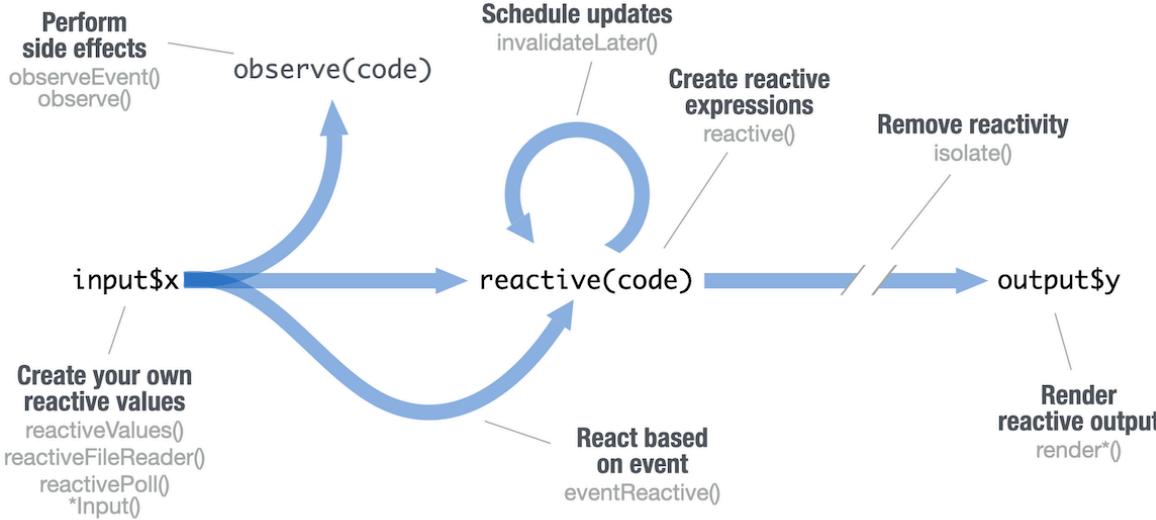
submitButton(text, icon, width)
(Prevent reactions for entire app)

textInput(inputId, label, value, width, placeholder) Also **textAreaInput()**

These are the core output types. See htmlwidgets.org for many more options.

Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error **Operation not allowed without an active reactive context**.



CREATE YOUR OWN REACTIVE VALUES

```
# *Input() example
ui <- fluidPage(
  textInput("a", "", "A")
)
```

```
#reactiveValues example
server <-
  function(input, output){
    rv <- reactiveValues()
    rv$number <- 5
  }
```

*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.

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)
```

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.

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)
```

eventReactive(eventExpr, valueExpr, event.env, event.quoted, value.env, value.quoted, ..., label, domain, ignoreNULL, ignoreInit)

Creates reactive expression with code in 2nd argument that only invalidates when reactive values in 1st argument change.

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.

UI

An app's UI is an HTML document.

Use Shiny's functions to assemble this HTML with R.

```
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

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>"))
```

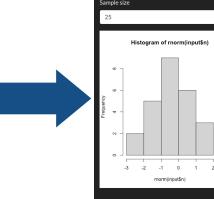
To include an image:

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.

Layouts

Combine multiple elements into a "single element" that has its own properties with a panel function, e.g.

```
wellPanel(
  dateInput("a", ""),
  submitButton()
)
```

```
absolutePanel()
conditionalPanel()
fixedPanel()
headerPanel()
inputPanel()
mainPanel()
```



```
navlistPanel()
sidebarPanel()
tabPanel()
tabsetPanel()
titlePanel()
wellPanel()
```

Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

sidebarLayout()

```
side panel main panel
```

fluidRow()

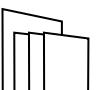
```
column col
column
```

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

```
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:



```
tab 1 tab 2 tab 3
contents
```

```
tab 1 tab 2 tab 3
contents
```

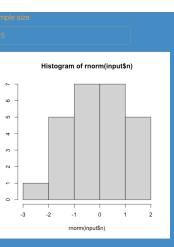
```
tab 1 tab 2 tab 3
contents
```

```
ui <- fluidPage( tabsetPanel(
  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.

Data & Variable Transformation with sjmisc Cheat Sheet



sjmisc complements dplyr, and helps with data transformation tasks and recoding *variables*.

sjmisc works together seamlessly with dplyr and pipes. All functions are designed to support labelled data.



Design Philosophy

The design of sjmisc functions follows the tidyverse-approach: first argument is always the data (either a *data frame* or *vector*), followed by variable names to be processed by the functions.

The returned object for each function *equals the type of the data-argument*.

Vector input

- If the data-argument is a *vector*, functions return a *vector*.



```
rec(mtcars$carb, rec = "1,2=1; 3,4=2; else=3")
```

Data frame input

- If the data-argument is a *data frame*, functions return a *data frame*.



```
rec(mtcars, carb, rec = "1,2=1; 3,4=2; else=3")
```

The ...-ellipses Argument

Apply functions to a single variable, selected variables or to a complete data frame.

Variable selection is powered by dplyr's `select()`: Separate variables with comma, or use dplyr's select-helpers to select variables, e.g. `?rec`:

```
rec(mtcars, one_of(c("gear", "carb")),
    rec = "min:3=1; 4:max=2")
```

```
rec(mtcars, gear, carb, rec = "min:3=1; 4:max=2")
```

Descriptives and Summaries

Most of the sjmisc functions (including recode-functions) also work on grouped data frames:

```
library(dplyr)
efc %>%
  group_by(e16sex, c172code) %>%
  frq(e42dep)
```

Frequency Tables

```
frq(x, ..., sort.frq = c("none", "asc", "desc"),
     weight.by = NULL, auto.grp = NULL, ...)
```

Print frequency tables of (labelled) vectors. Uses variable labels as table header.

```
data(efc); frq(efc, e42dep, c161sex)
```

Use this data set in examples!

```
flat_table(data, ..., margin = c("counts",
                                 "cell", "row", "col"), digits = 2,
            show.values = FALSE)
```

Print contingency tables of (labelled) vectors. Uses value labels.

```
flat_table(efc, e42dep, c172code, e16sex)
```

```
count_na(x, ...)
```

Print frequency table of tagged NA values.

```
library(haven); x <- labelled(c(1:3,
                                tagged_na("a", "a", "z")), labels =
                                c("Refused" = tagged_na("a"), "N/A" =
                                tagged_na("z")))
count_na(x)
```

Descriptive Summary

```
descr(x, ..., max.length = NULL)
```

Descriptive summary of data frames, including variable labels in output.

```
descr(efc, contains("cop"), max.length = 20)
```

Finding Variables in a Data Frame

Use `find_var()` to search for variables by names, value or variable labels. Returns vector/data frame.

```
# variables with "cop" in names and variable labels
find_var(efc, pattern = "cop", out = "df")
```

```
# variables with "level" in names and value labels
find_var(efc, "level", search = "name_value")
```

Recode and Transform Variables

Recode functions add a *suffix* to new variables, so original variables are preserved.

By default, original input data frame and new created variables are returned. Use `append = FALSE` to return the recoded variables only.

```
rec(x, ..., rec, as.num = TRUE, var.label =
     NULL, val.labels = NULL, append = TRUE,
     suffix = "_r")
```

Recode values, return result as numeric, character or categorical (factor).

```
rec(mtcars, carb, rec = "1,2=1; 3,4=2; else=3")
```

```
dicho(x, ..., dich.by = "median", as.num =
      FALSE, var.label = NULL, val.labels = NULL,
      append = TRUE, suffix = "_d")
```

Dichotomise variable by median, mean or specific value.

```
dicho(mtcars, disp)
```

```
split_var(x, ..., n, as.num = FALSE,
           val.labels = NULL, var.label = NULL,
           inclusive = FALSE, append = TRUE,
           suffix = "_g")
```

Split variable into equal sized groups. Unlike `dplyr::ntile()`, does not split original categories into different values (see examples in `?split_var`).

```
split_var(mtcars, mpg, disp, n = 3)
```

```
group_var(x, ..., size = 5, as.num = TRUE,
           right.interval = FALSE, n = 30, append =
           TRUE, suffix = "_gr")
```

Split variable into groups with equal value range, or into a max. # of groups (value range per group is adjusted to match # of groups).

```
group_var(mtcars, mpg, disp, size = 5)
```

```
group_var(mtcars, mpg, size = "auto", n = 4)
```

```
std(x, ..., robust = "sd", include.fac = FALSE,
     append = TRUE, suffix = "_z")
```

Z-standardise variables. Also `center()`.

```
std(efc, e17age, c160age)
```

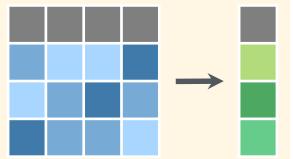
```
recode_to(x, ..., lowest = 0, highest = -1,
          append = TRUE, suffix = "_r0")
```

Shift ("renumber") categories or values.

```
recode_to(mtcars$gear)
```

Summarise Variables and Cases

The summary functions mostly mimic base R equivalents, but are designed to work together with pipes and dplyr.



```
row_sums(x, ..., na.rm = TRUE, var =
  "rowsums", append = FALSE)
```

Row sums of data frames.

```
row_sums(efc, c82cop1:c90cop9)
```

```
row_means(x, ..., n, var = "rowmeans",
           append = FALSE)
```

Row means, for at least `n` valid (non-NA) values.

```
row_means(efc, c82cop1:c90cop9, n = 7)
```

```
row_count(x, ..., count, var = "rowcount",
           append = FALSE)
```

Row-wise count # of values in data frames.

```
row_count(efc, c82cop1:c90cop9, count = 2)
```

Other Useful Functions

- `add_columns()` and `replace_columns()` to combine data frames, but either replace or preserve existing columns.

- `set_na()` and `replace_na()` to convert regular into missing values, or vice versa. `replace_na()` also replaces specific `tagged NA` values only.

- `remove_var()` and `var_rename()` to remove variables from data frames, or rename variables.

- `group_str()` to group similar string values. Useful for variables with similar, but not identically spelled string values that should be "merged".

- `merge_df()` to full join data frames and preserve value and variable labels.

- `to_long()` to gather multiple columns in data frames from wide into long format.

Use with %>% and dplyr

```
# use sjmisc-functions in pipes
mtcars %>% select(gear, carb) %>%
  rec(rec = "min:3=1; 4:max=2")
```

```
# use sjmisc-function inside mutate
mtcars %>% select(gear, carb) %>% mutate(
  carb2 = rec(carb, rec = "1,2=0;3:8=1"),
  gear2 = rec(gear, rec = "3=1;4:max=2"))
```



slackr :: CHEAT SHEET

Send files, messages, R objects, and images to Slack directly from R

Installation

CRAN version

```
install.packages("slackr")
```

Development version

```
devtools::install_github("mrkaye97/slackr")
```

Setup

- Go to <https://api.slack.com/apps>

- Click “Create New App” and then follow the setup instructions

Multi-channel bot

- Click “OAuth & Permissions” under “Features”
- Enable the following scopes in order to get all of the functionality:
 - channels:read
 - chat:write
 - users:read
 - chat:write.customize
 - files:read
 - chat:write.public
 - groups:read
 - im:write
 - groups:write
 - incoming-webhook
 - channels:history
- Click “Install to Workspace”
- Select a channel (for webhook messages)
- Copy the Bot User OAuth Access Token
- Click “Incoming Webhooks” under “Features”
- Copy the Webhook URL

Single-channel bot

- Click “Incoming “Webhooks” under “Features”
- Turn the “Activate Incoming Webhooks” switch on
- Click “Add New Webhook to Workspace”
- Select the channel you’d like the bot to post to
- Copy the Webhook URL

Setup environment variables for Slack API access

```
slackr_setup(
  channel = "#general",
  username = "slackr",
  icon_emoji = "",
  incoming_webhook_url = "",
  bot_user_oauth_token = "",
  config_file = "~/.slackr",
  echo = FALSE,
  cacheChannels = TRUE,
  cache_dir = "")
```

Generate Config Text (.txt) File (optional)
 bot_user_oauth_token: xoxb-[...]
 channel: #yourchannel
 username: yourusername
 incoming_webhook_url: https://hooks.slack.com/services/XXXXXX/XXXXX/XXXXXX

Common Functions

Post a ggplot to a Slack channel

```
ggslackr(
  plot = last_plot(),
  channels = Sys.getenv("SLACK_CHANNEL"),
  scale = 1,
  width = par("din")[1],
  height = par("din")[2],
  units = c("in", "cm", "mm"),
  dpi = 300,
  limitsize = TRUE,
  bot_user_oauth_token =
    Sys.getenv("SLACK_BOT_USER_OAUTH_TOKEN"),
  file = "ggplot",
  ...)
```

Save R objects to an RData file on Slack

```
save_slackr(
  ...,
  channels = Sys.getenv("SLACK_CHANNEL"),
  file = "slackr",
  bot_user_oauth_token =
    Sys.getenv("SLACK_BOT_USER_OAUTH_TOKEN"),
  plot_text = ""
)
```

Send result of R expressions to a Slack channel via webhook API

```
slackr_bot(
  ...,
  channel = "",
  username = "",
  icon_emoji = "",
  incoming_webhook_url =
    Sys.getenv("SLACK_INCOMING_URL_PREFIX"))
```

Delete the specified number of messages from the channel

```
slackr_delete(
  count,
  channel = Sys.getenv("SLACK_CHANNEL"),
  bot_user_oauth_token =
    Sys.getenv("SLACK_BOT_USER_OAUTH_TOKEN"))
```

Send a message to a Slack channel

```
slackr_msg(
  txt = "",
  channel = Sys.getenv("SLACK_CHANNEL"),
  username = Sys.getenv("SLACK_USERNAME"),
  icon_emoji = Sys.getenv("SLACK_ICON_EMOJI"),
  bot_user_oauth_token =
    Sys.getenv("SLACK_BOT_USER_OAUTH_TOKEN"),
  ...)
```

Send a file to Slack

```
slackr_upload(
  filename,
  title = basename(filename),
  initial_comment = basename(filename),
  channels = Sys.getenv("SLACK_CHANNEL"),
  bot_user_oauth_token =
    Sys.getenv("SLACK_BOT_USER_OAUTH_TOKEN"))
```

Other Usages

auth_test()

Checks authentication & identity against the Slack API

call_slack_api()

A wrapper function to call the Slack API with authentication and pagination

convert_response_to_tibble()

Convert Slack API json response to tibble

register_onexit()

Append text_slackr as on.exit to functions

slackr_census_fun()

Create a cache of the users and channels in the workspace in order to limit API requests

slackr_channels()

Get a data frame of Slack channels

slackr_chtrans()

Translate vector of channel names to channel IDs for API

slackr_dev()

Send the graphics contents of the current device to a Slack channel

slackr_history()

Reads history of a channel

slackr_ims()

Get a data frame of Slack IM IDs

slackr_users()

Get a data frame of Slack users

text_slackr()

Sends basic text to a slack channel

tex_slackr()

Post a tex output to a Slack channel

with_pagination()

Calls the slack API with pagination using cursors

Vignettes

Creating a single-channel bot

```
vignette('webhook-setup', package = 'slackr')
```

Creating a fully-functional multi-channel bot

```
vignette('scoped-bot-setup', package = 'slackr')
```

Usage

```
vignette('using-slackr', package = 'slackr')
```

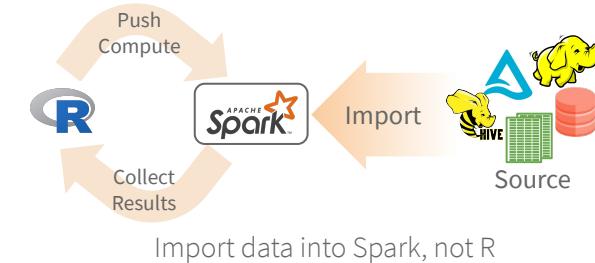
Data Science in Spark with sparklyr :: CHEAT SHEET



Intro

sparklyr is an R interface for Apache Spark™. **sparklyr** 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



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()`

HIVE TABLE `spark_read_table()`

ORC `spark_read_orc()`

LIBSVM `spark_read_libsvm()`

JDBC `spark_read_jdbc()`

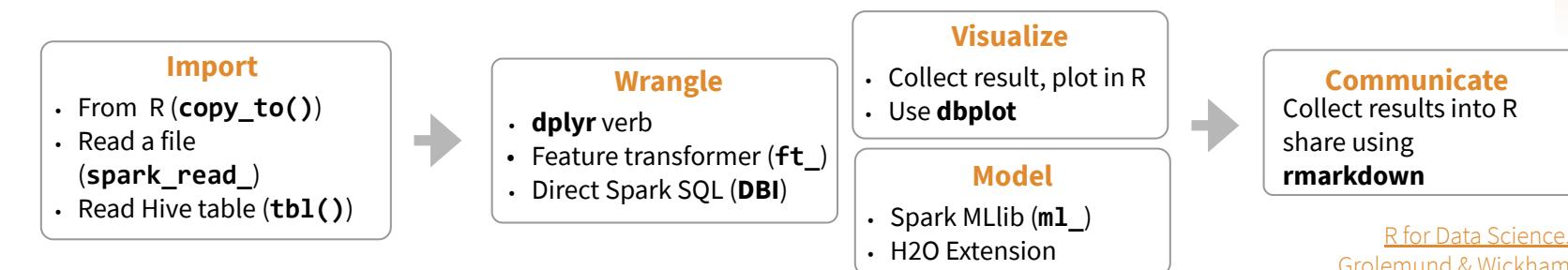
DELTA `spark_read_delta()`

R DATA FRAME INTO SPARK

`dplyr::copy_to(dest, df, name)`

FROM A TABLE IN 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)
  
```

FEATURE TRANSFORMERS

	<code>ft_max_abs_scaler()</code> - Rescale each feature individually to range [-1, 1]
	<code>ft_min_max_scaler()</code> - Rescale each feature individually to a common range [min, max] linearly
	<code>ft_ngram()</code> - Converts the input array of strings into an array of n-grams
	<code>ft_bucketed_random_projection_lsh()</code> <code>ft_minhash_lsh()</code> - Locality Sensitive Hashing functions for Euclidean distance and Jaccard distance (MinHash)
	<code>ft_normalizer()</code> - Normalize a vector to have unit norm using the given p-norm
	<code>ft_one_hot_encoder()</code> - Continuous to binary vectors
	<code>ft_pca()</code> - Project vectors to a lower dimensional space of top k principal components
	<code>ft_quantile_discretizer()</code> - Continuous to binned categorical values
	<code>ft_regex_tokenizer()</code> - Extracts tokens either by using the provided regex pattern to split the text
	<code>ft_standard_scaler()</code> - Removes the mean and scaling to unit variance using column summary statistics
	<code>ft_stop_words_remover()</code> - Filters out stop words from input
	<code>ft_string_indexer()</code> - Column of labels into a column of label indices.
	<code>ft_tokenizer()</code> - Converts to lowercase and then splits it by white spaces

Visualize

- Collect result, plot in R
- Use `dbplot`

Model

- Spark MLlib (`m1_`)
- H2O Extension

[R for Data Science](#), [Golem & Wickham](#)

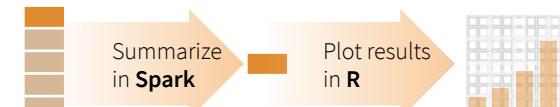
`ft_vectorAssembler()` - Combine vectors into single row-vector

`ft_vectorIndexer()` - Indexing categorical feature columns in a dataset of Vector

`ft_vectorSlicer()` - 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

```

copy_to(sc, mtcars) %>%
  group_by(cyl) %>%
  summarise(mpg_m = mean(mpg)) %>%
  collect() %>%
  ggplot() +
  geom_col(aes(cyl, mpg_m))
  
```

`copy_to` Summarize in Spark
`group_by` Collect results in R
`collect` Create plot

DBPLOT

```

copy_to(sc, mtcars) %>%
  dbplot_histogram(mpg) +
  labs(title = "Histogram of MPG")
  
```

`copy_to`
`dbplot_histogram` Calculates the histogram bins in Spark and plots in ggplot2
`dbplot_raster` Visualize 2 continuous variables. Use instead of `geom_point`

Data Science in Spark with sparklyr :: CHEAT SHEET



Modeling

REGRESSION

`ml_linear_regression()` - Regression using linear regression.

`ml_aft_survival_regression()` - Parametric survival regression model named accelerated failure time (AFT) model

`ml_generalized_linear_regression()` - Generalized linear regression model

`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()` - Naive Bayes Classifiers. 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(model,...)` | `ml_tree_feature_importance(model)` - 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()` - K-means clustering with support for k-means

FP GROWTH

`ml_fpgrowth()` | `ml_association_rules()` | `ml_freq_itemsets()` - A parallel FP-growth algorithm to mine frequent itemsets.

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

`correlate` package integrates with sparklyr

`copy_to(sc, mtcars) %>%
 correlate() %>%
 rplot()`

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.

UTILITIES

`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.

Start a Spark session

YARN CLIENT

1. Install RStudio Server on one of the existing nodes, preferably 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 (some base configurations included in the example)
`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. Install a local version of Spark:
`spark_install (version = "2.0.1")`
3. Open a connection
`spark_connect(master = "spark://host:port",
 version = "2.0.1",
 spark_home = spark_home_dir())`

LOCAL MODE

No cluster required. Use for learning purposes only

1. Install a local version of Spark:
`spark_install("2.3")`
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",
 version = "2.3.1"))`

MESOS

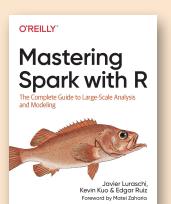
1. Install RStudio Server on one of the nodes
2. Open a connection

`sc <- spark_connect(master = "[Mesos URL]")`

CLOUD

- `Databricks` - `spark_connect(method = "databricks")`
`Qubole` - `spark_connect(method = "qubole")`

More Information



spark.rstudio.com

therinspark.com

Stata to R :: CHEAT SHEET



Introduction

This cheat sheet summarizes common Stata commands for econometric analysis and provides their equivalent expression in R.

References for importing/cleaning data, manipulating variables, and other basic commands include Hanck et al. (2019), [Econometrics with R](#), and Wickham and Grolemund (2017), [R for Data Science](#).

Example data comes from Wooldridge *Introductory Econometrics: A Modern Approach*. Download Stata data sets [here](#). R data sets can be accessed by installing the [`wooldridge` package](#) from CRAN.

All R commands written in base R, unless otherwise noted.

Setup

Note: While it is common to create a `log` file in Stata to store the commands and output of Stata sessions, the equivalent does not exist in R. A more savvy version in R is to create a [R-markdown](#) file to capture code and output.

```
ssc install outreg2 // install `outreg2` package. Note: unlike R packages, Stata packages do not have to be loaded each time once installed.
```

```
install.packages("wooldridge") # install `wooldridge` package  
  
data(package = "wooldridge") # list datasets in `wooldridge` package  
  
load(wage1) # load `wage1` dataset into session  
  
?wage1 # consult documentation on `wage1` dataset
```

Basic plots

example data: `wage1`

```
hist(wage) // histogram of `wage`  
hist(wage), by(nonwhite) //  
scatter(wage educ) // scatter plot of `wage` by `educ`  
twoway (scatter wage educ) (lfit wage educ) // scatter plot with fitted line  
graph box wage, by(nonwhite) // boxplot of wage by `nonwhite`
```

Summarize Data

example data: `wage1`

Where Stata only allows one to work with one data set at a time, multiple data sets can be loaded into the R environment simultaneously, and hence must be specified with each function call. Note: R does not have an equivalent to Stata's `codebook` command.

```
browse // open browser for loaded data  
  
describe // describe structure of loaded data  
summarize // display summary statistics for all variables in dataset  
list in 1/6 // display first 6 rows  
  
tabulate educ // tabulate `educ` variable frequencies  
tabulate educ female // cross-tabulate `educ` and `female` frequencies
```

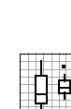


```
view(wage1) # open browser for loaded `wage1` data  
  
str(wage1) # describe structure of `wage1` data  
summary(wage1) # display summary statistics for `wage1` variables  
head(wage1) # display first 6 (default) rows data  
tail(wage1) # display last 6 rows  
  
table(wage1$educ) #tabulate `educ` frequencies  
table("yrs_edu" = wage1$educ, "female" = wage1$female) # tabulate `educ` frequencies name table columns
```

Tip: The {AER} package will automatically load other useful dependent packages, including: {car}, {lmtest}, {sandwich} which are used for many of the commands listed in this cheat sheet.



```
hist(wage1$wage) # histogram of `wage`  
  
{  
  plot(y = wage1$wage, x = wage1$educ) # scatter plot  
  abline(lm(wage1$wage~wage1$educ), col="red") # add fitted line to scatterplot  
  
  boxplot(wage1$wage~wage1$nonwhite) # boxplot of `wage` by `nonwhite`
```



Estimate Models, 1/2

OLS

example data: `wage1`

```
reg wage educ // simple regression of `wage` by `educ` (Results printed automatically).
```

```
reg wage educ if nonwhite==1 // add condition with if statement
```

```
reg wage educ exper, robust // multiple regression using HC1 robust standard errors
```

```
reg wage educ exper, cluster(numdep) // use clustered standard errors
```

Tip: An alternate way to compute robust standard errors in R for any models not covered by {estimatr} package is load the {AER} package and run:

```
coeftest(mod1, vcov. = vcovHC, type = "HC1")
```

MLE (Logit/Probit/Tobit)

example data: `mroz`

```
logit inlf nwifeinc educ // estimate logistic regression
```

```
probit inlf nwifeinc educ // estimate logistic regression
```

```
tobit hours nwifeinc educ, ll(0) // estimate tobit regression, lower-limit of y censored at zero
```

Postestimation, 1/2

example data: `wage1`

Note: Postestimation commands in Stata apply to the most recently run estimation commands.

```
reg wage educ // estimation used for the following post-estimation commands
```

```
predict yhat // get predicted values from last estimation, store as `yhat`
```

```
predict e, res // get residuals from last estimation, store as `e`
```

```
mod1 <- lm(wage ~ educ, data = wage1) # simple regression of `wage` by `educ`, store results in mod1  
summary(mod1) # print summary of `mod1` results
```

```
mod2 <- lm(wage ~ educ, data = wage1[wage1$nonwhite==1, ]) # add condition with if statement
```

```
mod3 <- estimatr::lm_robust(wage ~ educ + exper, data = wage1, se_type = "stata") # multiple regression with HC1 (Stata default) robust standard errors, use {estimatr} package
```

```
mod4 <- estimatr::lm_robust(wage ~ educ + exper, data = wage1, clusters = numdep) # use clustered standard errors.
```

```
mod_log <- glm(inlf~nwifeinc + educ + family=binomial(link="logit"), data=mroz) # estimate logistic regression
```

```
mod_pro <- glm(inlf~nwifeinc + educ + family=binomial(link="probit"), data=mroz) # estimate logistic regression
```

```
mod_tob <- AER::tobit(hours ~ nwifeinc + educ, left = 0, data = mroz) # estimate tobit regression, lower-limit of y censored at zero, use {AER} package
```

```
mod1 <- lm(wage ~ educ, data = wage1) # estimation used for the following post-estimation commands  
yhat <- predict(mod1) # get predicted values
```

```
e <- residuals(mod1) # get residual values
```



Create/Edit Variables

example data: `wage1`

Note: where Stata only allows one to work with one data set at a time, multiple data sets can be loaded into the R environment simultaneously, hence the data set must be specified for each command.

```
gen exper2 = exper^2 // create
`exper` squared variable
egen wage_avg = mean(wage) // create
average wage variable

drop tenursq // drop `tenursq`
variable

keep wage educ exper nonwhite // keep
selected variables
```

```
tab numdep, gen(numdep) // create
dummy variables for `numdep`

recode exper (1/20 = 1 "1 to 20
years") (21/40 = 2 "21 to 40 years")
(41/max = 3 "41+ years"),
gen(experlvl) // recode `exper` and
gen new variable
```

Statistical tests / diagnostics

```
reg lwage educ exper // estimation
used for examples below
estat hettest // Breusch-Pagan /
Cook-Weisberg test for
heteroskedasticity
estat ovtest // Ramsey RESET test
for omitted variables
ttest wage, by(nonwhite) //
independent group t-test, compare
means of same variable between
groups
```

Interactions, categorical/continuous variables

In Stata, it is common to use special operators to specify the treatment of variables as continuous (`c.`) or categorical(`i.`). Similarly, the `#` operator denotes different ways to return the interaction of those variables. Here we show some common uses of these operators as well as their R equivalents.

```
reg lwage i.numdep // treat
`numdep` as a factor variable
reg lwage c.educ#c.exper // return
interaction term only
reg lwage c.educ##c.exper // return
full factorial specification
reg lwage c.exper##i.numdep // return full,
interact continuous and categorical
```

```
wage1$exper2 <- wage1$exper^2 # create `exper` squared variable
wage1$wage_avg <- mean(wage1$wage) # create average wage variable

wage1$tenursq <- NULL #drop `tenursq`

wage1 <- wage1[, c("wage", "educ",
"exper", "nonwhite")] # keep selected
variables

wage1 <-
fastDummies::dummy_cols(wage1,
select_columns = "numdep") # create
dummy variables for `numdep`, use
{fastDummies} package

wage1$experlvl <- 3 # recode `exper`
wage1$experlvl[wage1$exper < 41] <- 2
wage1$experlvl[wage1$exper < 21] <- 1
```

example data: `wage1`

```
mod <- lm(lwage ~ educ exper, data =
wage1) # estimate used for examples
below
lmtest::bptest(mod) # Breusch-Pagan /
Cook-Weisberg test for hetero-
skedasticity using the {lmtest}
package
lmtest::resettest(mod) # Ramsey
RESET test
t.test(wage ~ nonwhite, data =
wage1) # independent group t-test
```

example data: `wage1`

```
lm(lwage ~ as.factor(numdep), data
= wage1) # treat `numdep` as factor
lm(lwage ~ educ:exper, data =
wage1) # return interaction term
only
lm(lwage ~ educ*exper, data =
wage1) # return full factorial
specification
lm(wage ~ exper*as.factor(numdep),
data = wage1) # return full,
interact continuous and categorical
```

Estimate Models, 2/2

Panel/Longitudinal

example data: `murder`

```
xtset id year // set `id` as
entities (panel) and `year` as
time variable
xtdescribe // describe pattern of
xt data
xtsum // summarize xt data
xtreg mrd rte unem, fe // fixed
effects regression
```



Instrumental Variables (2SLS)

example data: `mroz`

```
ivreg lwage (educ = fatheduc),
first // show results of first
stage regression
est first // test IV and
endogenous variable
ivreg lwage(educ = fatheduc) //
show results of 2SLS directly
```



```
plm::is.pbalanced(murder$id,
murder$year) # check panel balance
with {plm} package
modfe <- plm::plm(mrd rte ~ unem,
index = c("id", "year"), model =
"within", data = murder) # estimate
fixed effects ("within") model
summary(modfe) # display results
```

```
modiv <- AER::ivreg(lwage ~ educ |
fatheduc, data = mroz) # estimate
2SLS with {AER} package
summary(modiv, diagnostics = TRUE)
# get diagnostic tests of IV and
endogenous variable
```

Post-estimation, 2/2

example data: `wage1`

Note: Postestimation commands in Stata apply to the most recently run estimation commands.

```
reg lwage educ exper##exper // estimation
used for following post-
estimation commands
estimates store mod1 // stores in
memory the last estimation results
to `mod1'
```

```
margins // get average predictive
margins
margins, dydx(*) // get average
marginal effects for all variables
marginsplot // plot marginal
effects
```

```
margins, dydx(exper) // average
marginal effects of experience
margins, at(exper=(1(10)51)) // average
predictive margins over
`exper` range at 10-year increments
```

```
estimates use mod1 // loads `mod1`
back into working memory
estimates table mod1 mod2 // display
table with stored
estimation results
```



```
margins::prediction(mod1) # get
average predictive margins with
{margins} package
m1 <- margins::margins(mod1) # get
average marginal effects for all
variables
plot(m) # plot marginal effects
```

```
summary(m) # get detailed summary of
marginal effects
margins::prediction(mod1, at =
list(exper = seq(1,51,10))) #
predictive margins over `exper` range
at 10-year increments
```

```
stargazer::stargazer(mod1, mod2, type
= "text") # use {stargazer} package,
with `type=text` to display results
within R. Note: `type` also can be
changed for LaTeX and HTML output.
```



String manipulation with stringr :: CHEAT SHEET



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")

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"))

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", "-")
	str_to_lower(string, locale = "en")¹ Convert strings to lower case. str_to_lower(sentences)
	str_to_upper(string, locale = "en")¹ Convert strings to upper case. str_to_upper(sentences)
	str_to_title(string, locale = "en")¹ Convert strings to title case. Also str_to_sentence() . str_to_title(sentences)

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

	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.

Need to Know

Pattern arguments in string are interpreted as regular expressions *after any special characters have been parsed*.

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes ("") or single quotes('').

Some characters cannot be represented directly in an R string. These must be represented as **special characters**, sequences of characters that have a specific meaning., e.g.

Special Character	Represents
\\\	\
'"	"
\n	new line

Run `?"""` to see a complete list

Because of this, whenever a \ appears in a regular expression, you must write it as \\ in the string that represents the regular expression.

Use `writeLines()` to see how R views your string after all special characters have been parsed.

```
writeLines("|\.")  
# \.
```

```
writeLines("\| is a backslash")  
# \| is a backslash
```

INTERPRETATION

Patterns in stringr are interpreted as regexs. To change this default, wrap the pattern in one of:

`regex(pattern, ignore_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...)`
Modifies a regex to ignore cases, match end of lines as well of end of strings, allow R comments within regex's , and/or to have . match everything including \n.
`str_detect("i", regex("i", TRUE))`

`fixed()` Matches raw bytes but will miss some characters that can be represented in multiple ways (fast). `str_detect("\u0130", fixed("i"))`

`coll()` Matches raw bytes and will use locale specific collation rules to recognize characters that can be represented in multiple ways (slow). `str_detect("\u0130", coll("i", TRUE, locale = "tr"))`

`boundary()` Matches boundaries between characters, line_breaks, sentences, or words. `str_split(sentences, boundary("word"))`

Regular Expressions -

Regular expressions, or *regexp*s, are a concise language for describing patterns in strings.

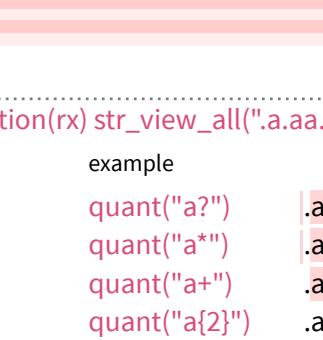
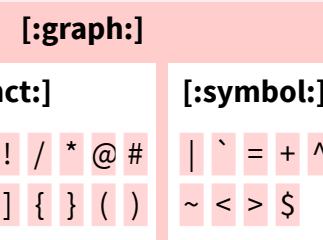
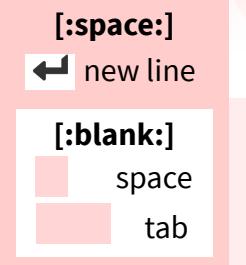
MATCH CHARACTERS

string (type this)	regexp (to mean this)	matches (which matches this)	example
	a (etc.)	a (etc.)	see("a")
\\.	\\.	.	see("\\.")
\\!	\\!	!	see("\\!")
\\?	\\?	?	see("\\?")
\\\\	\\\\	\	see("\\\\")
\\(\\((see("\\()")
\\)	\\))	see("\\)")
\\{	\\{	{	see("\\{")
\\}	\\}	}	see("\\}")
\\n	\\n	new line (return)	see("\\n")
\\t	\\t	tab	see("\\t")
\\s	\\s	any whitespace (S for non-whitespaces)	see("\\s")
\\d	\\d	any digit (D for non-digits)	see("\\d")
\\w	\\w	any word character (W for non-word chars)	see("\\w")
\\b	\\b	word boundaries	see("\\b")
	[:digit:] ¹	digits	see("[:digit:]")
	[:alpha:] ¹	letters	see("[:alpha:]")
	[:lower:] ¹	lowercase letters	see("[:lower:]")
	[:upper:] ¹	uppercase letters	see("[:upper:]")
	[:alnum:] ¹	letters and numbers	see("[:alnum:]")
	[:punct:] ¹	punctuation	see("[:punct:]")
	[:graph:] ¹	letters, numbers, and punctuation	see("[:graph:]")
	[:space:] ¹	space characters (i.e. \s)	see("[:space:]")
	[:blank:] ¹	space and tab (but not new line)	see("[:blank:]")
.	.	every character except a new line	see(".")

see <- function(rx) str_view_all("abc ABC 123\\t.!?\n\\n", rx)

abc ABC 123 .!?\n
abc ABC 123 .!?

see <- function(rx) str_view_all("abc ABC 123\\t.!?\n\\n", rx)



ALTERNATES

regexp	matches	example

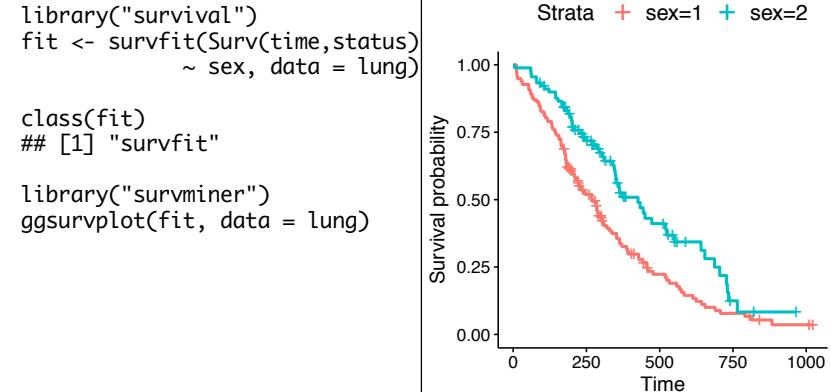
<tbl_r cells="3" ix="1" maxcspan="1" maxrspan="1"

Creating Survival Plots

Informative and Elegant with survminer

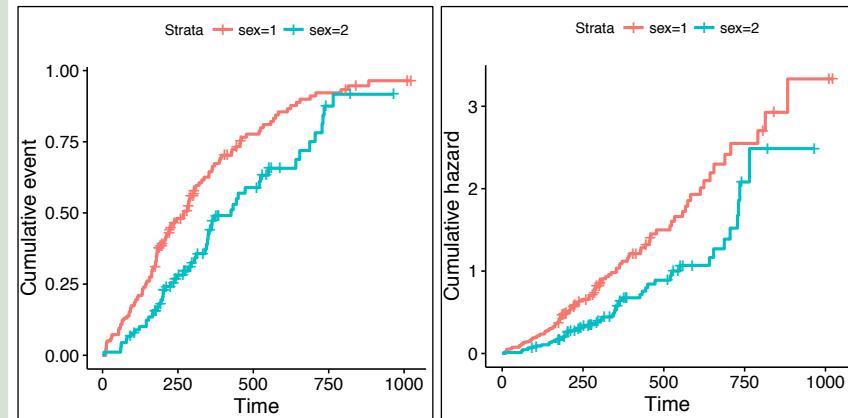
Survival Curves

The **ggsurvplot()** function creates **ggplot2** plots from **survfit** objects.



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")
```



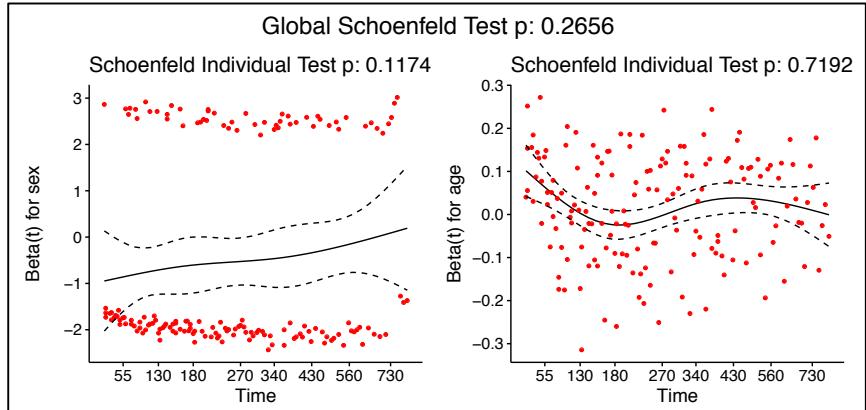
With lots of graphical parameters you have full control over look and feel of the survival plots; position and content of the legend; additional annotations like p-value, title, subtitle.

```
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"))
```

Diagnostics of Cox Model

The function **cox.zph()** from **survival** package may be used to test the proportional hazards assumption for a Cox regression model fit. The graphical verification of this assumption may be performed with the function **ggcooxzph()** 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")
ggcooxzph(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".

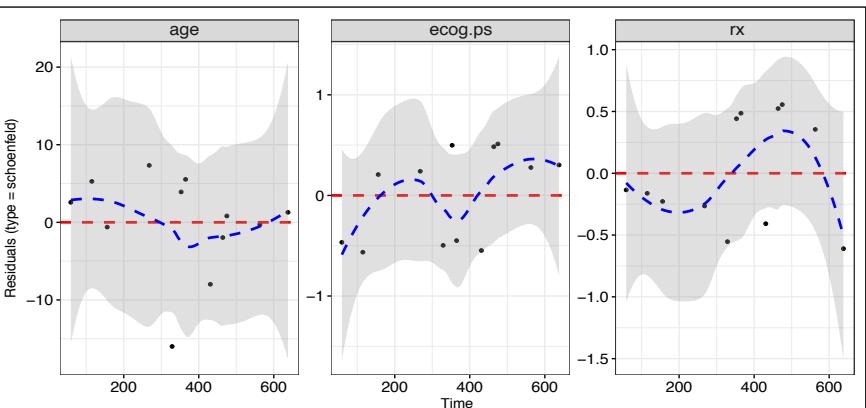
The **ox.scale** argument defines what shall be plotted on the OX axis. Possible values are "linear.predictions", "observation.id", "time".

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")
```



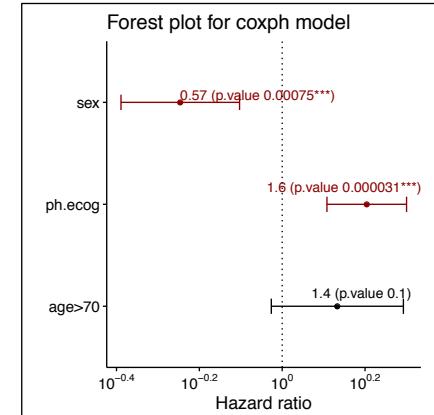
Summary of Cox Model

The function **ggforest()** from the **survminer** package creates a forest plot 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
```

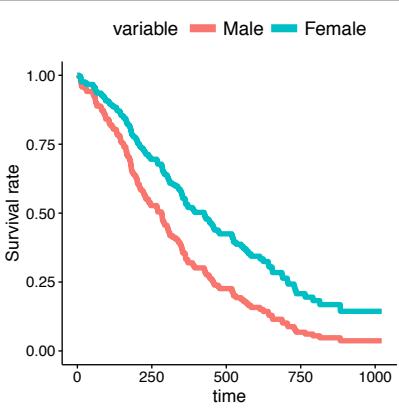
```
ggforest(fit)
```



The function **ggcoxaadjustedcurves()** 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.

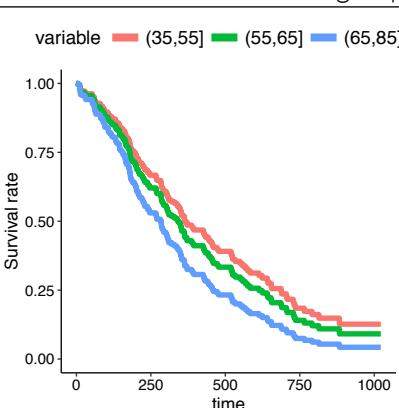
Note that these curves differ from Kaplan Meier estimates since they present expected survival based on given Cox model.

```
library("survival")
library("survminer")
lung$sex <- ifelse(lung$sex == 1,
"Male", "Female")
fit <- coxph(Surv(time, status) ~ sex + ph.ecog + age,
data = lung)
ggcoxaadjustedcurves(fit, data=lung,
variable=lung$sex)
```



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))
ggcoxaadjustedcurves(fit, data=lung,
variable=lung$age3)
```



R Syntax Comparison :: CHEAT SHEET

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))
```

R Syntax Comparison :: CHEAT SHEET

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`, `tidyverse`, 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)
```

The teachR's :: CHEAT SHEET

Getting ready to teach some R? Use our cheat sheet to **prepare, teach and debrief**



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

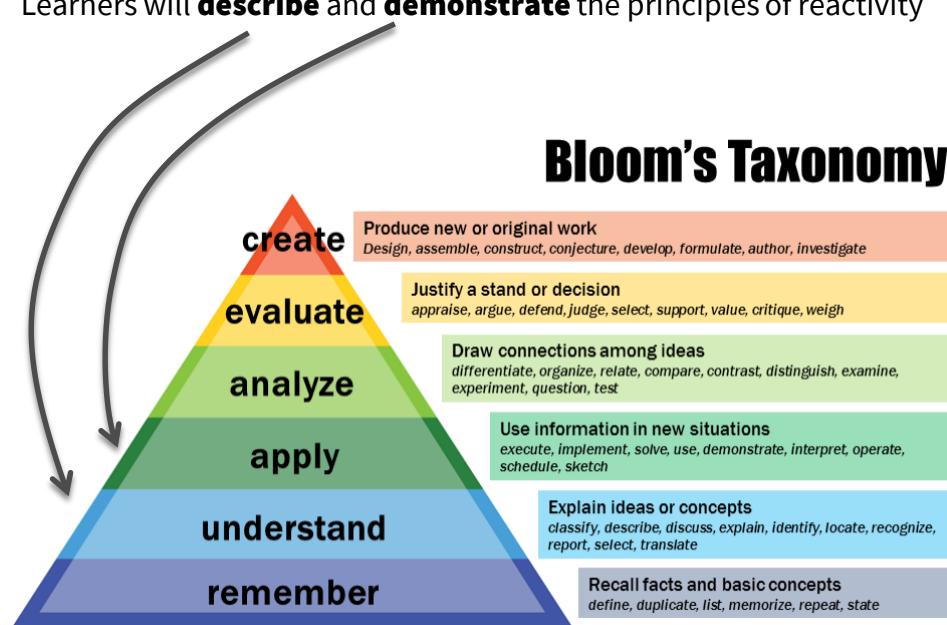


For example (R shiny - novice):

Learners will **describe** and **demonstrate** the principles of reactivity

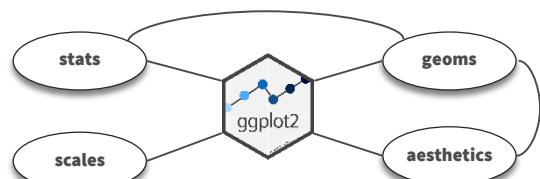


Bloom's Taxonomy



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_ *() +`

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

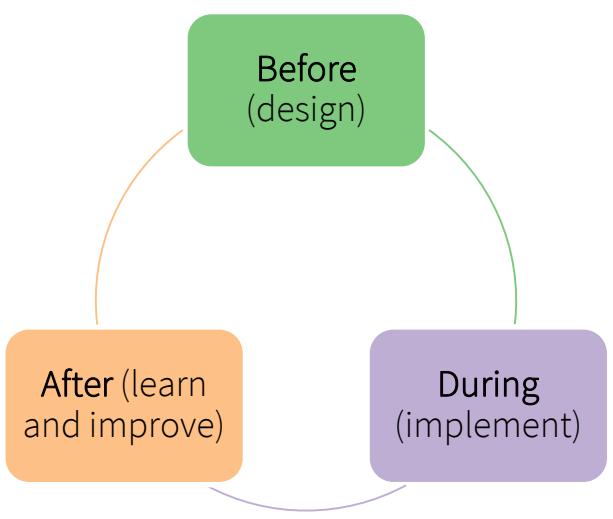
(r4ds.had.co.nz)

Advanced R / Hadley Wickham (adv-r.had.co.nz)

RStudio Cheat sheets:

<https://www.rstudio.com/resources/cheatsheets/>

Iterative work flow



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)

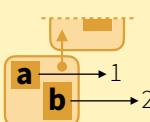
Tidy evaluation with rlang :: CHEAT SHEET



Vocabulary

Tidy Evaluation (Tidy Eval) is not a package, but a framework for doing non-standard evaluation (i.e. delayed evaluation) that makes it easier to program with tidyverse functions.

pi



Symbol - a name that represents a value or object stored in R. `is_symbol(expr(pi))`

Environment - a list-like object that binds symbols (names) to objects stored in memory. Each env contains a link to a second, **parent** env, which creates a chain, or search path, of environments. `is_environment(current_env())`

`rlang::caller_env(n = 1)` Returns calling env of the function it is in.

`rlang::child_env(.parent, ...)` Creates new env as child of .parent. Also **env**.

`rlang::current_env()` Returns execution env of the function it is in.

1

abs (1)

Constant - a bare value (i.e. an atomic vector of length 1). `is_bare_atomic(1)`

Call object - a vector of symbols/constants/calls that begins with a function name, possibly followed by arguments. `is_call(expr(abs(1)))`

pi — code
3.14 — result

Code - a sequence of symbols/constants/calls that will return a result if evaluated. Code can be:

1. Evaluated immediately (**Standard Eval**)
 2. Quoted to use later (**Non-Standard Eval**)
- `is_expression(expr(pi))`

e
a + b
q
a + b, [a b]

Expression - an object that stores quoted code without evaluating it. `is_expression(expr(a + b))`

Quosure- an object that stores both quoted code (without evaluating it) and the code's environment. `is_quosure(quo(a + b))`

[a b] `rlang::quo_get_env(quo)` Return the environment of a quosure.

[a b] `rlang::quo_set_env(quo, expr)` Set the environment of a quosure.

a + b `rlang::quo_get_expr(quo)` Return the expression of a quosure.

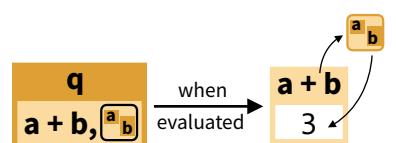
Expression Vector - a list of pieces of quoted code created by base R's `expression` and `parse` functions. Not to be confused with **expression**.

R Studio

Quoting Code

Quote code in one of two ways (if in doubt use a quosure):

QUOSURES



Quosure- An expression that has been saved with an environment (aka a closure).

A quosure can be evaluated later in the stored environment to return a predictable result.

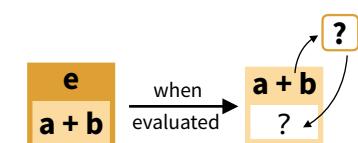
`rlang::quo(expr)` Quote contents as a quosure. Also **quos** to quote multiple expressions. `a <- 1; b <- 2; q <- quo(a + b); qs <- quos(a, b)`

`rlang::enquo(arg)` Call from within a function to quote what the user passed to an argument as a quosure. Also **enquos** for multiple args.
`quote_this <- function(x) enquo(x)`
`quote_these <- function(...) enquos(...)`

`rlang::new_quosure(expr, env = caller_env())` Build a quosure from a quoted expression and an environment.
`new_quosure(expr(a + b), current_env())`



EXPRESSION



Quoted Expression - An expression that has been saved by itself.

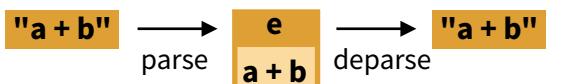
A quoted expression can be evaluated later to return a result that will depend on the environment it is evaluated in

`rlang::expr(expr)` Quote contents. Also **exprs** to quote multiple expressions. `a <- 1; b <- 2; e <- expr(a + b); es <- exprs(a, b, a + b)`

`rlang::enexpr(arg)` Call from within a function to quote what the user passed to an argument. Also **enexprs** to quote multiple arguments.
`quote_that <- function(x) enexpr(x)`
`quote_those <- function(...) enexprs(...)`

`rlang::ensym(x)` Call from within a function to quote what the user passed to an argument as a symbol, accepts strings. Also **ensyms**.
`quote_name <- function(name) ensym(name)`
`quote_names <- function(...) ensyms(...)`

Parsing and Deparsing



Parse - Convert a string to a saved expression.

• • •

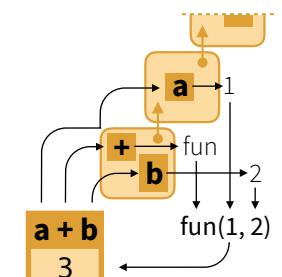
`rlang::parse_expr(x)` Convert a string to an expression. Also **parse_exprs**, **sym**, **parse_quo**, **parse_quos**. `e <- parse_expr("a+b")`

Deparse - Convert a saved expression to a string.

• • •

`rlang::expr_text(expr, width = 60L, nlines = Inf)` Convert expr to a string. Also **quo_name**. `expr_text(e)`

Evaluation



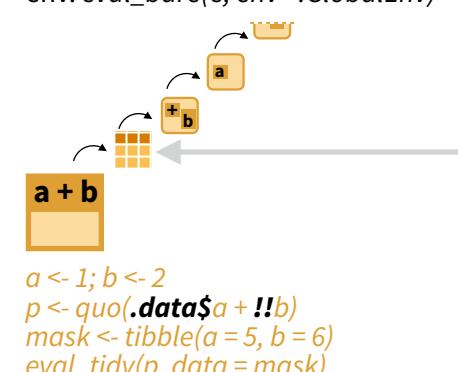
To evaluate an expression, R :

1. Looks up the symbols in the expression in the active environment (or a supplied one), followed by the environment's parents
2. Executes the calls in the expression

The result of an expression depends on which environment it is evaluated in.

QUOTED EXPRESSION

`rlang::eval_bare(expr, env = parent.frame())` Evaluate expr in env. `eval_bare(e, env = GlobalEnv)`



QUOSURES (and quoted exprs)

`rlang::eval_tidy(expr, data = NULL, env = caller_env())` Evaluate expr in env, using data as a **data mask**. Will evaluate quosures in their stored environment. `eval_tidy(q)`

Data Mask - If data is non-NULL, `eval_tidy` inserts data into the search path before env, matching symbols to names in data.

Use the pronoun **.data\$** to force a symbol to be matched in data, and **!!** (see back) to force a symbol to be matched in the environments.

Building Calls

`rlang::call2(fn, ..., .ns = NULL)` Create a call from a function and a list of args. Use **exec** to create and then evaluate the call. (See back page for !!!) `args <- list(x = 4, base = 2)`

log (x = **4**, base = **2**)

2

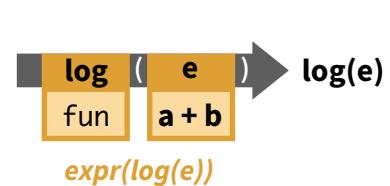
```
call2("log", x = 4, base = 2)
call2("log", !!!args)
exec("log", x = 4, base = 2)
exec("log", !!!args)
```



Quasiquotation (!!, !!!, :=)

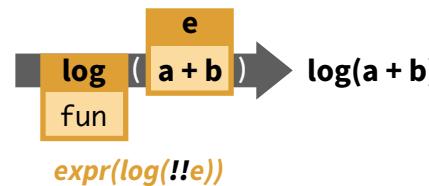
QUOTATION

Storing an expression without evaluating it.
e <- expr(a + b)



QUASIQUOTATION

Quoting *some* parts of an expression while evaluating and then inserting the results of others (**unquoting** others).
e <- expr(a + b)

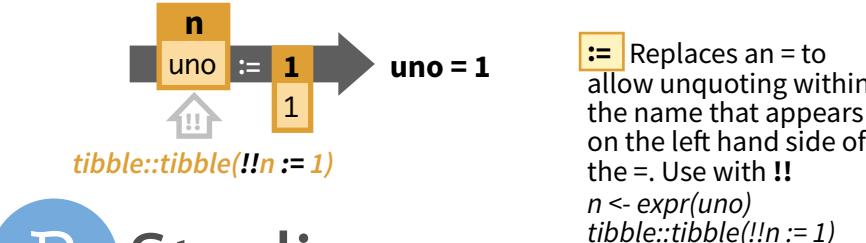
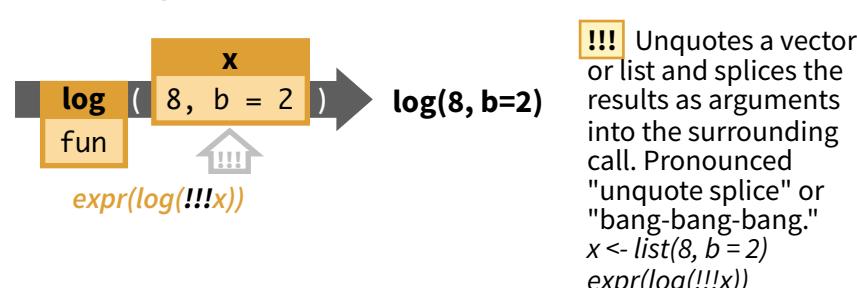
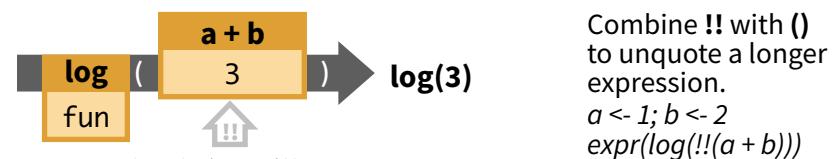
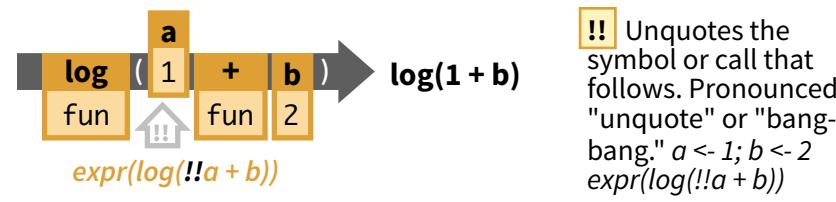


rlang provides !!, !!!, and := for doing quasiquotation.

!!, !!!, and := are not functions but syntax (symbols recognized by the functions they are passed to). Compare this to how

- . is used by magrittr::%>%()
- . is used by stats::lm()
- .x is used by purrr::map(), and so on.

!!, !!!, and := are only recognized by some rlang functions and functions that use those functions (such as tidyverse functions).



Programming Recipes

Quoting function- A function that quotes any of its arguments internally for delayed evaluation in a chosen environment. You must take **special steps to program safely** with a quoting function.

How to spot a quoting function?
A function quotes an argument if the argument returns an error when run on its own.

Many tidyverse functions are quoting functions: e.g. **filter**, **select**, **mutate**, **summarise**, etc.

```
dplyr::filter(cars, speed == 25)
      speed dist
      1     25    85
```

```
speed == 25
      Error!
```

PROGRAM WITH A QUOTING FUNCTION

```
data_mean <- function(data, var) {
  require(dplyr)
  var <- rlang::enquo(var) 1
  data %>%
    summarise(mean = mean(!!var)) 2
}
```

1. Capture user argument that will be quoted with rlang::enquo.
2. Unquote the user argument into the quoting function with !!.

PASS MULTIPLE ARGUMENTS TO A QUOTING FUNCTION

```
group_mean <- function(data, var, ...) {
  require(dplyr)
  var <- rlang::enquo(var)
  group_vars <- rlang::enquos(...) 1
  data %>%
    group_by(!!!group_vars) %>%
    summarise(mean = mean(!!var)) 2
}
```

1. Capture user arguments that will be quoted with rlang::enquos.
2. Unquote splice the user arguments into the quoting function with !!!.

MODIFY USER ARGUMENTS

```
my_do <- function(f, v, df) {
  f <- rlang::enquo(f)
  v <- rlang::enquo(v)
  todo <- rlang::quo(!!f)(!!v)) 2
  rlang::eval_tidy(todo, df) 3
}
```

1. Capture user arguments with rlang::enquo.
2. **Unquote** user arguments into a new expression or quoture to use
3. **Evaluate** the new expression/ quoture instead of the original argument

APPLY AN ARGUMENT TO A DATA FRAME

```
subset2 <- function(df, rows) {
  rows <- rlang::enquo(rows) 1
  vals <- rlang::eval_tidy(rows, data = df)
  df[vals, , drop = FALSE] 2
}
```

1. Capture user argument with rlang::enquo.
2. Evaluate the argument with rlang::eval_tidy. Pass the data frame to **data** to use as a data mask.
3. **Suggest** in your documentation that your users use the **.data** and **.env** pronouns.

WRITE A FUNCTION THAT RECOGNIZES QUASIQUOTATION (!!,,:=)

1. Capture the quasiquotation-aware argument with rlang::enquo.
2. Evaluate the arg with rlang::eval_tidy.

```
add1 <- function(x) {
  q <- rlang::enquo(x)
  rlang::eval_tidy(q) + 1
}
```

1
2

PASS TO ARGUMENT NAMES OF A QUOTING FUNCTION

```
named_m <- function(data, var, name) {
  require(dplyr)
  var <- rlang::enquo(var)
  name <- rlang::ensym(name) 1
  data %>%
    summarise (!!name := mean(!!var)) 2
}
```

1. Capture user argument that will be quoted with rlang::ensym.
2. Unquote the name into the quoting function with !! and :=.

PASS CRAN CHECK

```
#' @importFrom rlang .data
  mutate_y <- function(df) {
    dplyr::mutate(df, y = .data$a + 1)
  }
```

1
2

Quoted arguments in tidyverse functions can trigger an **R CMD check** NOTE about undefined global variables. To avoid this:

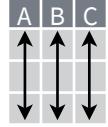
1. Import rlang::.data to your package, perhaps with the roxygen2 tag **@importFrom rlang .data**
2. Use the **.data\$** pronoun in front of variable names in tidyverse functions

Data tidying with `tidyr` :: CHEAT SHEET

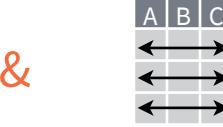


Tidy data is a way to organize tabular data in a consistent data structure across packages.

A table is tidy if:



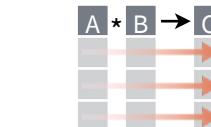
Each **variable** is in its own **column**



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



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` <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

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	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	

x	x1	x2	x3
A	1	3	
B	1	4	
B	2	3	
B	2	3	NA

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)`

Handle Missing Values

Drop or replace explicit missing values (NA).

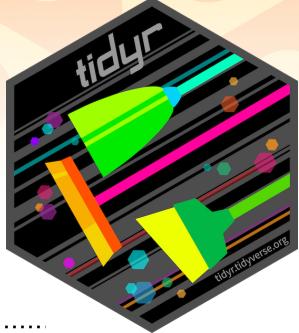
x	x1	x2
A	1	
B	NA	
C	NA	
D	3	
E	NA	

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



Nested Data

A **nested data frame** stores individual tables as a list-column of data frames within a larger organizing data frame. List-columns can also be lists of vectors or lists of varying data types.

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
Amy	1975	27.5	-79.0
Amy	1975	28.5	-79.0
Amy	1975	29.5	-79.0
Bob	1979	22.0	-96.0
Bob	1979	22.5	-95.3
Bob	1979	23.0	-94.6
Zeta	2005	23.9	-35.6
Zeta	2005	24.2	-36.1
Zeta	2005	24.7	-36.6

name	yr	lat	long
Amy	1975	27.5	-79.0
Amy	1975	28.5	-79.0
Amy	1975	29.5	-79.0
Bob	1979	22.0	-96.0
Bob	1979	22.5	-95.3
Bob	1979	23.0	-94.6

name	data
Luke	<tibble [50x3]>
C-3PO	<tibble [50x3]>
R2-D2	<tibble [50x3]>

name	yr	lat	long
Amy	2005	23.9	-35.6
Amy	2005	24.2	-36.1
Amy	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.

```
tribble(~max, ~seq,
       3, 1:3,
       4, 1:4,
       5, 1:5)
```

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.

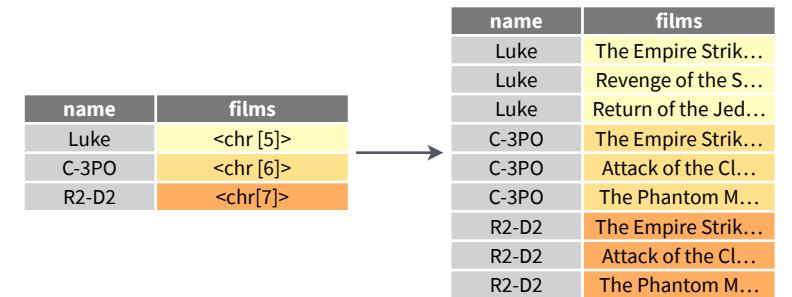
```
mtcars %>%
  group_by(cyl) %>%
  summarise(q = list(quantile(mpg)))
```

RESHAPE NESTED DATA

unnest(data, cols, ..., keep_empty = FALSE) Flatten nested columns back to regular columns. The inverse of `nest()`.
`n_storms %>% unnest(data)`

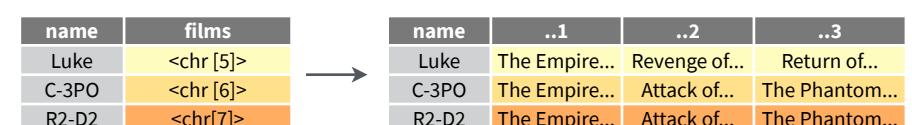
unnest_longer(data, col, values_to = NULL, indices_to = NULL)
Turn each element of a list-column into a row.

```
starwars %>%
  select(name, films) %>%
  unnest_longer(films)
```



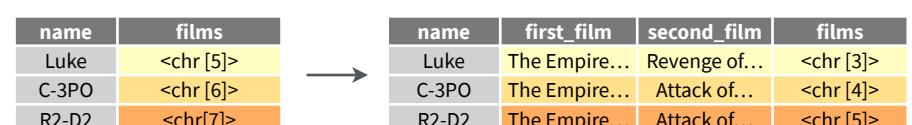
unnest_wider(data, col) Turn each element of a list-column into a regular column.

```
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.

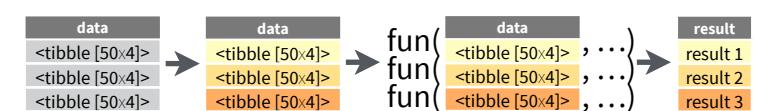
```
starwars %>%
  select(name, films) %>%
  hoist(films, first_film = 1, second_film = 2)
```



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)))`

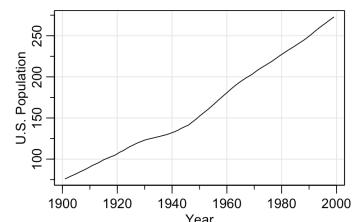
`length()` returns one integer per row

See **purrr** package for more list functions.

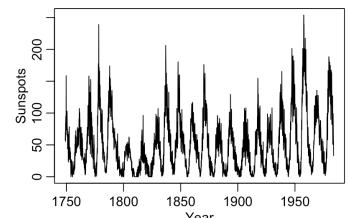
Time Series Cheat Sheet

Plot Time Series

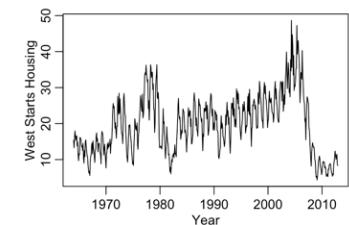
1. `tsplot(x=time, y=data)`



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

$$X_t = Z_t + \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \dots + \theta_q Z_{t-q}$$

ARMA (p, q)

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \dots + \phi_p X_{t-p} + Z_t + \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \dots + \theta_q Z_{t-q}$$

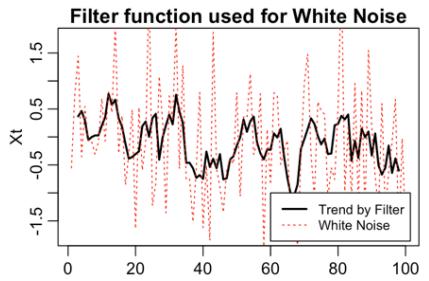
Simulation of ARMA (p, q)

```
arima.sim(model=list(ar=c(phi1, ..., phi_p),
                     ma=c(theta1, ..., theta_q)), n=n)
```

Filters

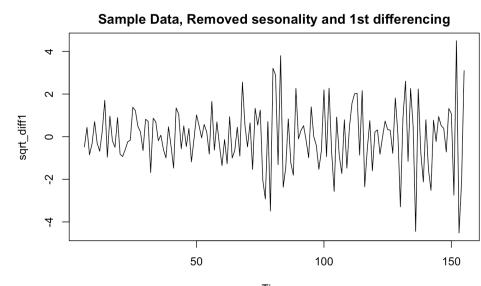
Linear Filter: `filter()`

```
filter(data, filter=filter_coefficients, sides=2,
       method="convolution", circular=F)
```



Differencing Filter: `diff()`

```
diff(data, lag=4, differences=1)
```

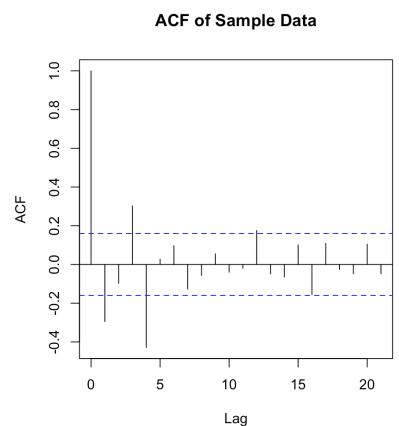


Auto-correlation

Use ACF and PACF to detect model

(Complete) Auto-correlation function: `acf()`

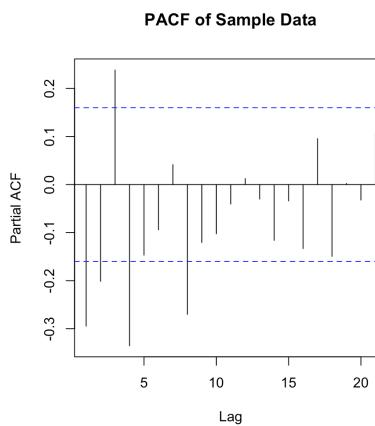
```
acf(data, type='correlation', na.action=na.pass)
```



Partial Auto-correlation function: `pacf()`

```
pacf(data, na.action=na.pass)
```

OR: `acf(data, type='partial', na.action=na.pass)`



Forecasting

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)
```

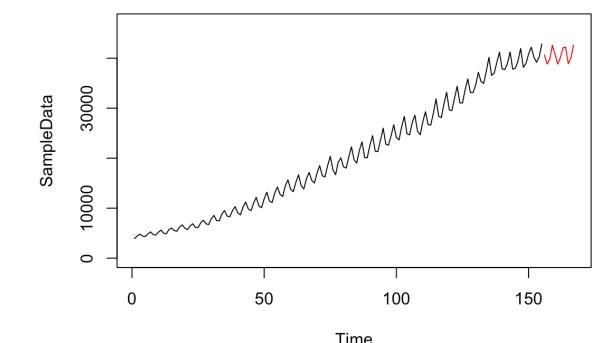
```
ts.plot(data,
```

```
       xlim=c(1, length(data)+number_to_predict),
```

```
       ylim=c(0, max(fit$pred+1.96*fit$se)))
```

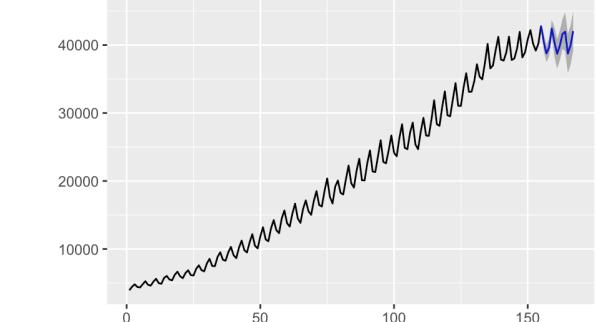
```
lines(length(data)+1:length(data)+
```

```
       number_to_predict, fit$pred)
```



OR: `autoplot(forecast(arima_model, level=c(95), h=number_to_predict))`

Predicted value and Conf Interval of ARIMA





Class Agnostic Time Series with tsbox :: CHEAT SHEET

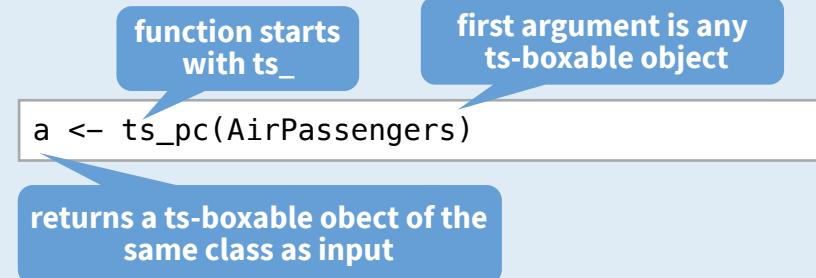
Basics

IDEA

tsbox provides a time series toolkit which:

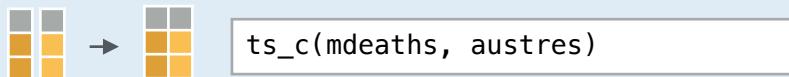
1. works identically with most time series **classes**
2. handles regular and irregular **frequencies**
3. **converts** between classes and frequencies

Most functions in tsbox have the same structure:

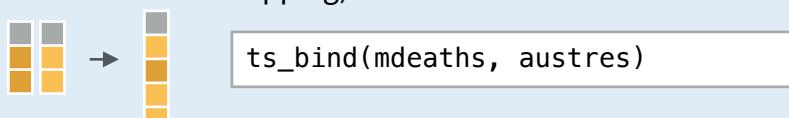


COMBINE TIME SERIES

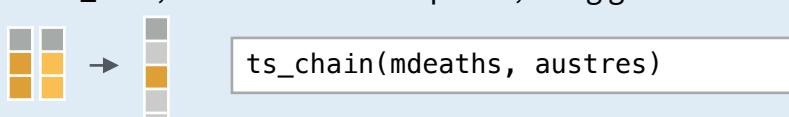
collect time series of **all classes** and **frequencies** as multiple time series



combine time series to a new, single time series (first series wins if overlapping)

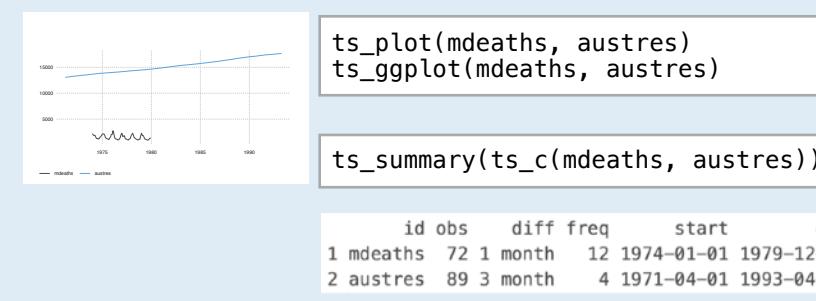


like `ts_bind`, but extra- and retropolate, using growth rates



PLOT AND SUMMARIZE

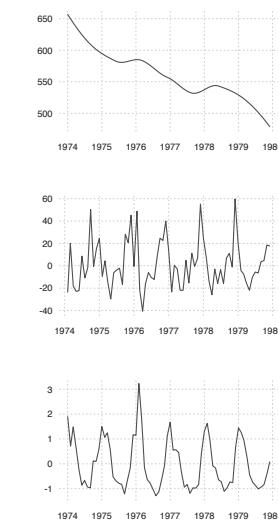
Plot time series of **all classes** and **frequencies**



Helper Functions

Transform time series of **all classes** and **frequencies**

TRANSFORM

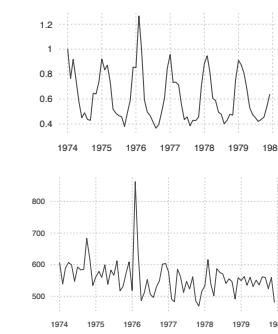


`ts_trend()`: Trend estimation based on loess

`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)`



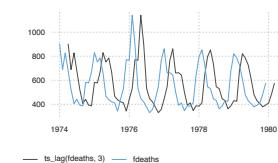
`ts_index()`: Index, based on levels

`ts_compound()`: Index, based on growth rates

`ts_index(fdeaths, base = 1976)`

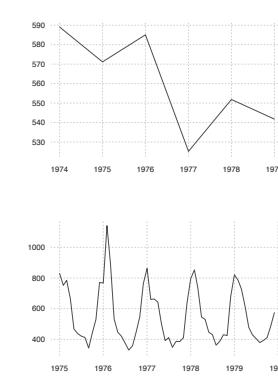
`ts_seas()`: seasonal adjustment using X-13
`ts_seas(fdeaths)`

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

Cheat Sheet :: VEGAN



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 = "")

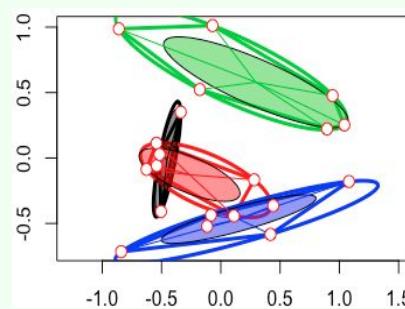
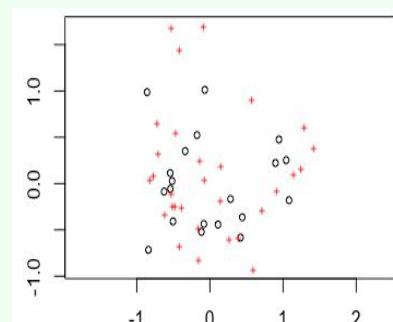
type = "p" results with points of black circles to indicate sites and red pluses to show species

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

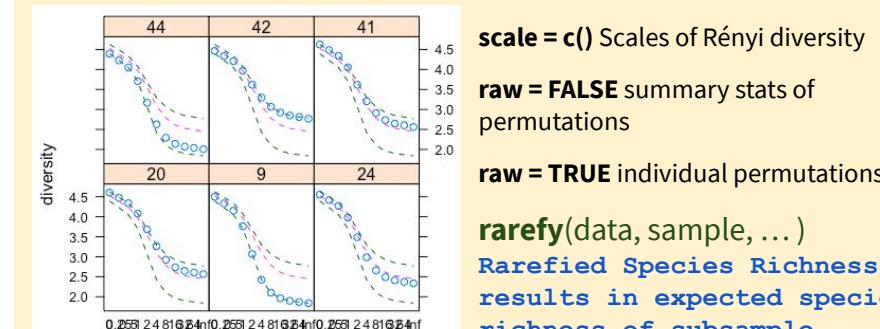
permutations = control values, or permutation index

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

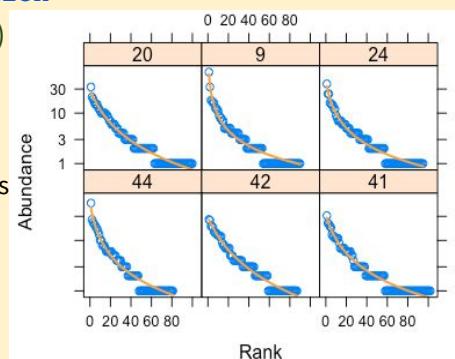
radfit(data, ...) Fits the most popular model to data using maximum likelihood estimation

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

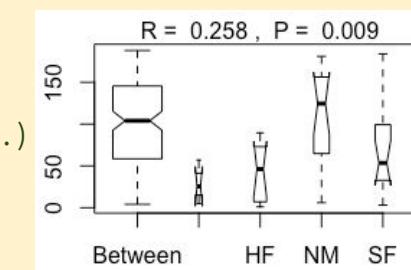
method = "" can specify which beta index to use (24 options)

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

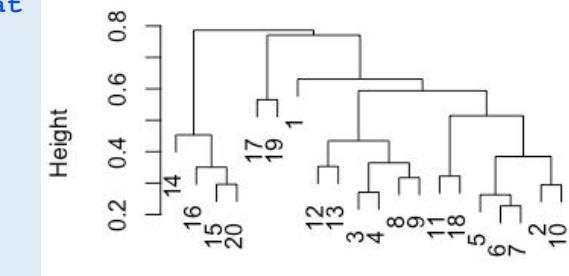
vegdist(data, method = "", na.rm = FALSE, ...) Dissimilarity indices

method = "dissimilarity index"

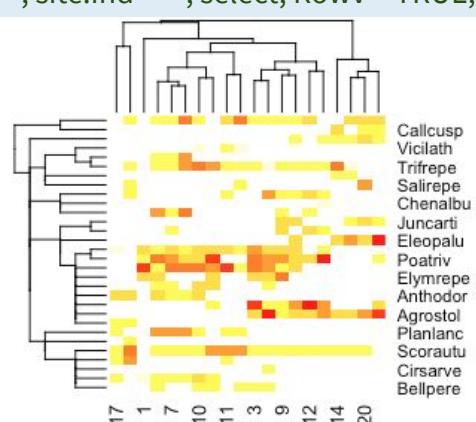
- "manhattan", "euclidean", "canberra", "clark", "bray", "kulczynski", "jaccard", "gower", "altGower", "morisita", "horn", "mountford", "raup", "binomial", "chao", "cao" or "mahalanobis".

Other Fun Features

vegemitte(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

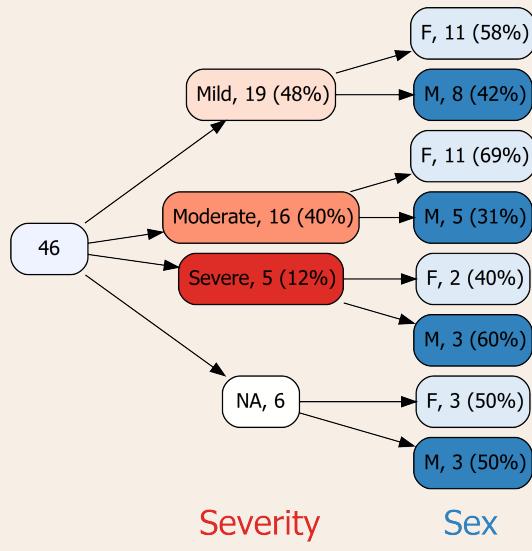
reference = data to be used to compare for joint analysis

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)`



Severity

Sex

Pruning

Parameter	Effect
<code>prune</code>	Remove identified nodes and their descendants.
<code>keep</code>	Only retain identified nodes and their descendants.
<code>prunebelow</code>	Remove descendants of identified nodes.
<code>follow</code>	Only retain descendants of identified nodes.

Example: `prune=list(Severity=c("Moderate","Severe"))`

Labels

Parameter setting	
<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>varnamepointsize=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>imagedwidth="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	
<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

<code>vtree(FakeData, "Severity_Sex", pattern=T, varnamebold=T)</code>	<code>vtree(FakeData, "Severity_Sex", ptable=T)</code>																																				
	<table border="1"> <thead> <tr> <th>n</th> <th>pct</th> <th>Severity</th> <th>Sex</th> </tr> </thead> <tbody> <tr> <td>2</td> <td>4</td> <td>Severe</td> <td>F</td> </tr> <tr> <td>3</td> <td>7</td> <td><NA></td> <td>F</td> </tr> <tr> <td>3</td> <td>7</td> <td><NA></td> <td>M</td> </tr> <tr> <td>3</td> <td>7</td> <td>Severe</td> <td>M</td> </tr> <tr> <td>5</td> <td>11</td> <td>Moderate</td> <td>M</td> </tr> <tr> <td>8</td> <td>17</td> <td>Mild</td> <td>M</td> </tr> <tr> <td>11</td> <td>24</td> <td>Mild</td> <td>F</td> </tr> <tr> <td>11</td> <td>24</td> <td>Moderate</td> <td>F</td> </tr> </tbody> </table>	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
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																																		
<code>pattern</code>	<code>Severity</code>																																				
<code>Severity</code>	<code>Sex</code>																																				

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 R² 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.

`<package>`

2 A `<package>` block combines all functions from the same package.

`<function>`

3 Within a `<function>` block, explanations/interpretations for the function as such or for specific elements of the return object can be provided.

`<result>`

4 Packages explanations/ interpretations related to one element of the function's return object.

```

<xplain>
  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. Level=1 explanations will also be presented when Level=2 or Level=3 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 !**! 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"
 "items"

Calling xplain()

1	<code>call</code>	Call of the explained function as string
	<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")`