

```
#####
# R #
#####

# install.packages("<package name>")
# library(<package name>)

# print(<something>)
# Assignment statement: x <- <something>

# Manipulating objects in the workspace:
# ls()           # list all objects in memory
# rm(<o1>, <o2>, <o3>, ...) # remove one or more objects from memory by their names
# rm(list = ls()) # remove all objects from memory (usually not recommended)

# Operators:
# +           Add, 2 + 3 = 5
# -           Subtract, 5 - 2 = 3
# *           Multiply, 2 * 3 = 6
# /           Divide, 6 / 2 = 3
# ^           Exponent, 2 ^ 3 = 8
# %%          Modulus operator, 9%%2 = 1
# %/%         Integer division, 9 %/% 2 = 4
# <           Less than
# >           Greater than
# =           Equal to
# <=          Less than or equal to
# >=          Greater than or equal to
# !=          Not equal to
# !           Not
# |           OR
# &           And

# Expressions:
# <x> / <y> - <z>^2 ...

# Absolute value:
# abs(<value>)

# Vectors:
# <y> <- c(<something1>, <something2>, <something3>, ...)
# <y> <- rep(<something>, <times>)
# <y> <- <int1>:<int2>
# <y> <- seq(from = <value1>, to = <value2>, by = <step>)

# Matrices:
# <m> <- matrix(c(3, 5, 7, 1, 9, 4), nrow = 3, ncol = 2, byrow = TRUE)
# <m>.nrow <- nrow(<m>) # number of rows
# <m>.ncol <- ncol(<m>) # number of columns
# <m> <- t(<m>)         # transpose <m>
# <m>[3,2]
# <m>[2, ]

# Lists: ordered collections of elements of different types
# <list> <- list(<e1.name> = <e1>, <e2.name> = <e2>, <e3.name> = <e3>, ...)
# <list>[[<index>]]    # accessing list element by index, showing value only (returns a vector)
# <list>[<index>]     # accessing list element by index, showing both name and value (returns a list)
# <list>$<element.name> # accessing list element by its name
# is.list(<something>) # Is <something> a list?
```

```

# <combined.list> <- c(<list1>, <list2>, <list3>, ...) # list concatenation
# names(<list>) # names of list elements
# <list>[names(<list>) == <element.name>] # all elements of a list having the same name
# unlist(<list>) # convert list into a named vector
# unlist(<list>, use.names = FALSE) # convert list into a character vector
# append(<list>, # insert new element into an existing list, after index <n>
#     list(<el.name> = <e>), # new element must be a list itself, that's why list(<el.name> = <e>)
#     <n>) # <n> is optional; if omitted, new element is appended at the end
# <list>[[<n>]] <- NULL # remove <n>th element from <list>

# class(<something>) # data type
# mode(<something>), typeof(<something>) # how a data item is internally stored in memory

# Factors:
# b <- c(1, 2, 2, 2, 3, 1, 1, 4, 5, 4)
# b.as.factor <- as.factor(b)
# levels(b.as.factor)
# f <- factor(c(1, 2, 3))

# Dataframes:
# e.g., <dataframe> <- as.data.frame(<matrix>)
# str(<dataframe>)

# Reading a dataset:
# <dataframe> <- read.csv("<filename>", stringsAsFactors = FALSE)

# Saving a dataset (modified or newly created dataset):
# write.csv(x = <dataframe>, file = "<filename>", row.names = F) # do not include the row names (row numbers) column
# saveRDS(object = <dataframe or another R object>, file = "<filename>") # save R object for the next session
# <dataframe or another R object> <- readRDS(file = "<filename>") # restore R object in the next session

# Examining a dataframe:
# str(<dataframe>) # structure of <dataframe>, all variables/columns
# dim(<dataframe>) # showing dimensions (numbers of rows and columns) of a dataframe
# names(<dataframe>) # showing column names
# head(<dataframe>) # the first few rows
# tail(<dataframe>) # the last few rows
# <dataframe>[ , ] # the entire dataframe
# <dataframe> # the entire dataframe
# <dataframe>[<m>, ] # m-th row
# <dataframe>[ , <n>] # n-th column
# summary(<dataframe>$<column>) # summarizing a variable/column values
# fix(<dataframe>) # editing a dataframe
# new.df <- edit(<dataframe>) # editing a dataframe and assigning the modified dataframe to another dataframe

# Adding/Removing columns to/from a dataframe:
# <dataframe>$<new column name> <- <default value> # adding a new column (default values)
# <dataframe>$<column name> <- NULL # removing a column

# Adding a new row to a dataframe - the row must be a 1-line dataframe with the same column names:
# <new row> <- data.frame(<column name 1> = <value 1>, <column name 2> = <value 2>,...)
# <new data frame> <- rbind(<dataframe>, <new row>) # append new row to the end of the existing dataframe
# <new data frame> <- rbind(<dataframe>[1:i, ], # insert new row in the middle
#     <new row>,
#     <dataframe>[(i + 1):nrow(<dataframe>), ])

# Removing rows from a dataframe
# <dataframe>[-i, ] # show dataframe without i-th row
# <dataframe>[-c(i, j, k), ] # show dataframe without rows i, j, k

```

```

# <dataframe> <- <dataframe>[-i, ] # remove i-th row from dataframe
# <dataframe> <- <dataframe>[-c(i, j, k), ] # remove rows i, j, k from dataframe
# <dataframe> <- <dataframe>[-(i:k), ] # remove rows i to k from dataframe

# Changing column names:
# colnames(<dataframe>)[i] <- "<new name>"

# Changing row names:
# rownames(<dataframe>)[i] <- "<new name>"
# rownames(<dataframe>) <- c("<new name 1>", "<new name 2>",...)
# rownames(<dataframe>) <- c(1, 2,...)
# rownames(<dataframe>) <- list("<new name 1>", <numeric 2>,...)

# Slicing and dicing dataframes:
# <selection> <- <dataframe>[<some rows>, <some columns>]
# <selection> <- <dataframe>[i:k, c("<column 1>", "<column 2>",...)]
# <selection> <- <dataframe>[<indexes>, ]
# <selection> <- subset(<dataframe>, # subset() is much like SELECT... FROM... WHERE
# <logical condition for the rows to return>,
# <select statement for the columns to return>) # can be omitted; column names not prefixed by <dataframe>$
# <new dataframe> <- <dataframe>[, c("<col1 name>", "<col2 name>")]
# <new dataframe> <- <dataframe>[, <col1 index>:<col2 index>]]

# Shuffling rows/columns:
# <dataframe> <- <dataframe>[sample(nrow(<dataframe>)), ] # shuffle row-wise
# <dataframe> <- <dataframe>[, sample(ncol(<dataframe>))] # shuffle column-wise

# Replacing selected values in a column:
# <selected var name> <- <dataframe>$<column> == <selected value>
# <dataframe>$<column>[<selected var name>] <- <new value>

# Applying functions to all elements in rows/columns of a dataframe:
# apply(<dataframe>, <1 | 2>, <function(x) {...}>) # 1 | 2: apply function(x) by row | column
# IMPORTANT: use drop = FALSE in apply(...) when subsetting <dataframe> with [],
# i.e. <dataframe>[i, j, drop = FALSE]
# sapply(<vector>, FUN = function(x) {...}) # function(x): function to be applied to each element of <vector>

# Partitioning a dataframe:
# install.packages('caret')
# library(caret)
# set.seed(<any specific int>) # allows for repeating the randomization process exactly
# <indexes> <- createDataPartition(<dataframe>$<column>, p = 0.8, list = FALSE)
# <partition 1> <- <dataframe>[<indexes>, ]
# <partition 2> <- <dataframe>[-<indexes>, ]

# for, if, break, next:
# for (<i> in <int vector>) {
# <line 1>
# <line 2>
# ...
# if (<logical condition>) {
# <line i1>
# <line i2>
# ...
# break # break: exit the loop; next: skip the remaining lines in this iteration
# }
# ...
# <line n>
# }

```

```

# while, if-else, break, next:
# <i> <- <initial value>
# while (logical condition involving <i>) {
#   <line 1>
#   <line 2>
#   ...
#   if (<logical condition>) {
#     <line i1>
#     <line i2>
#     ...
#     break      # break: exit the loop; next: skip the remaining lines in this iteration
#   } else {
#     <line j1>
#     <line j2>
#     ...
#   }
#   ...
#   <line n>
#   <i> <- <modify <i>>
# }

# ifelse(<condition>, v1, v2)  # can return a vector

# Data type conversion
# b <- c(1, 2, 2, 2, 3, 1, 1, 4, 5, 4)
# b.as.factor <- as.factor(b)
# levels(b.as.factor)
# e.g., <dataframe> <- as.data.frame(<matrix>)
# str(<dataframe>)
# ...
# Convert numeric to factor:
# <dataframe>$<numeric column with few different values> <-
#   factor(<dataframe>$<numeric column with few different values>,
#         levels = c(0, 1, ..., k), labels = c("<l1>", "<l2>", ..., "<lk>"))

# Attributes of R objects (dataframes, matrices, factors, lists, tables...)
# attributes(<dataframe> | <matrix> | <factor> | <list> | table | ...)

# Tables
# The table() function:
# table(<var>) # typically a factor or an integer var
# The prop.table() function:
# prop.table(table(<var>))
# round(prop.table(table(<var>)), digits = <n>)
# Row and column margins:
# table(<var1>, <var2>) # <var1>, <var2>: usually factors or integers
# table(<rows title> = <var1>, <columns title> = <var2>) # add common titles for rows/columns
# prop.table(table(<var1>, <var2>), margin = 1) # all row margins (sums of values by row) are 1.0
# prop.table(table(<var1>, <var2>), margin = 2) # all column margins (sums of values by column) are 1.0

# Vectors
# Differences in initializing vectors and dataframe columns:
# <vector> <- rep(<value>, <times>)
# <vector> <- <value>
# <dataframe>$<column> <- rep(<value>, <times>)
# <dataframe>$<column> <- <value>
# Length of a vector:
# length(<vector>)

```

```

# Counting the number of elements with the values of <x> in a vector:
# 1. <table> <- table(<vector>)
#     <table>
#     <table>["<x>"], or <table>[names(<table>) == <x>]
# 2. sum(<vector> == <x>)
# 3. length(which(<vector> == <x>)) # which() is like WHERE in SQL
# Appending an element to a vector:
# <vector> <- c(<vector>, <element>) # type conversion occurs if <element> is of different type than v[i]
# <vector> <- append(<vector>, <element>) # type conversion occurs if <element> is of different type than v[i]
# <vector> <- append(<vector>, <element>,
#                   after = <n>) # insert <=> append at a desired location
# <vector> <- append(<vector>, NA)
# Removing NAs from a vector in NA-sensitive functions:
# <function>(<vector>, na.rm = TRUE)
# Selecting items matching criteria from a numeric vector (added check for NAs and NaNs):
# <numeric vector> <- c(<n1>, <n2>, <n3>, ..., NA, ...NaN)
# <selected> <- <numeric vector>[<logical criterion> & !is.na(<numeric vector>)] # is.na() is TRUE for both NA and NaN
# is.na() is the only way to test if <something> is NA (<something> == NA does not work)
# Range of a numeric vector:
# range(<vector>)
# Create numeric vector with <length> elements:
# <vector> <- vector(mode = "numeric", length = <length>)

# Check if numeric variables follow normal distribution:
# summary(<numeric variable>) # the mean and the median values similar: probably normal distribution
# plot(density((<numeric variable>)) # visual inspection
# hist(<numeric variable>, breaks = <n>) # visual inspection; <n>: number of bins in the histogram
# qqnorm(<numeric variable>) # values lie more or less along the diagonal (straight line)
# shapiro.test(<numeric variable>) # good for small sample sizes, e.g. n < ~2000; H0: normal distribution

# Discretizing numeric variables (using bnlearn::discretize()):
# library(bnlearn)
# ?discretize()
# <new dataframe with discretized variables> <-
#   discretize(<numeric dataframe>, # <original dataframe>[, c(<num. col. 1>, <num. col. 1>, ...)
#             method = "quantile" | # use equal-frequency intervals (default)
#             method = "interval", # use equal-length intervals
#             breaks = c(<n1>, <n2>, ..., <ncol>)) # no. of discrete intervals for each column

# Scatterplot matrices (useful for examining the presence of linear relationship between several pairs of variables):
# pairs(~<x1> + <x2> + ..., data = <dataframe>)

# Data normalization:
# library(clusterSim)
# <dataframe with numeric columns> <- # works with vectors and matrices as well
#   data.Normalization(<dataframe with numeric columns>,
#                     type = "n4", # normalization: (x - min(x)) / (max(x) - min(x))
#                     normalization = "column") # normalization by columns
# Alternatively:
# <norm.f> = function(x) {(x-min(x))/(max(x)-min(x))}
# <dataframe with numeric columns>[] <- # [] preserves the "data.frame" class
#   lapply(<dataframe with numeric columns>, <norm.f>)
# Alternatively:
# install.packages("scales")
# library(scales)
# <dataframe with numeric columns> <-
#   lapply(<dataframe with numeric columns>, rescale) # normalization: (x - min(x)) / (max(x) - min(x))
# Alternatively:
# install.packages("caret")

```

```

# library(caret)
# <pre-processed object> <-
#   preprocess(<dataframe with numeric columns>,
#             method = 'range') # normalization: (x - min(x)) / (max(x) - min(x))
# <dataframe with numeric columns> <-
#   predict(<pre-processed object>,
#          <dataframe with numeric columns>)

# Correlation plots: # correlations between numeric variables in the dataset
# <numeric dataframe> <- # create all-numeric dataframe,
#   data.frame(<num col 1 name> = <dataframe>$<num col 1>, # leave out all non-numeric columns
#             <num col 2 name> = <dataframe>$<num col 2>, # from the original dataframe
#             ...)
# <correlation matrix> <- cor(<numeric dataframe>) # all-numeric dataframe
# library(corrplot)
# corrplot.mixed(<correlation matrix>, tl.cex = <text font size>, number.cex = <number font size>)

# Quantiles/Percentiles:
# <quantiles> <- quantile(<dataset>$<column name>, # examine the 0th, 2.5th, ..., percentile
#                       probs = seq(from = 0.0, to = 0.1, by = 0.025))

# Sorting:
# sort(<numeric vector>) # sort <numeric vector>

# install.packages("knitr") # pretty-printing tables etc. in the console
# library(knitr) # (a set of "fancy" reporting tools)
# kable(x = <stats>, format = "rst")

#### ggplot2

# Bar graphs:
# ggplot(data = <dataframe>,
#        aes(x = <column 1>, y = <column 2>, fill = <column 1>)) + # fill = <column 1> is optional; no y for counts
#   geom_bar(stat = "identity") + # "identity" for values, "count" for counts
#   xlab("<x-axis label>") + ylab("<y-axis label>") +
#   ggtitle("<graph title>")
# Render a bar chart that shows mean values on the y axis (not sums of y values):
# ggplot(data = <dataframe>,
#        aes(x = <column 1>, y = <column 2>, fill = <column 1>)) + # fill = <column 1> is optional; no y for counts
#   geom_bar(stat = "summary", fun.y = "mean") # use both stat = "summary" and fun.y = "mean"
# ggplot(<dataframe>, aes(x = <column 1>, fill = <column 2>)) +
#   geom_bar(position = "dodge", width = <bin width>) + # "dodge": bar grouping, <bin width>: 0.2-0.6
#   labs(x = "<x-label>", y = "<y-label>", title = "<title>") +
#   theme_bw()

# Line graphs:
# ggplot(data = <dataframe>,
#        aes(x = <column 1>, y = <column 2>, group = 1)) + # group = 1: one line, all points connected
#   geom_line(colour = "<colour>", linetype = "<linetype>", size = <line thickness>) +
#   geom_point(colour = "<colour>", size = <point size>, shape = <point shape>, fill = "<point fill colour>") +
#   xlab("<x-axis label>") + ylab("<y-axis label>") +
#   ggtitle("<graph title>")
# All parameters in geom_line() and in geom_point() are optional.
# The defaults are: colour = "black", linetype = "solid", size = 1, shape = 21 (circle), fill = "black"
# See http://www.cookbook-r.com/Graphs/Colors\_\(ggplot2\)/
# for more information on colors.
# See http://www.cookbook-r.com/Graphs/Shapes\_and\_line\_types/
# for information on shapes and line types.

```

```

# Scatterplots:
# ggplot(<dataset>, aes(x = <num.var.1>, y = <num.var.2>)) +
#   geom_point(shape = <n>,          # <n> = 1: hollow circle
#             fill = <color 1>,     # color of point fill (optional)
#             color = <color 2>,    # color of point line (optional)
#             size = <s>) +         # size of point line (optional)
#   geom_smooth(method = lm,        # add regression line (optional); if left out, nonlinear best-fit line is shown
#              se=FALSE)          # do NOT show 95% confidence region as a shaded area (optional)
# <scatterplot> <-
#   ggplot(<dataset>, aes(x = <num.var.1>, y = <num.var.2>)) +
#   geom_point(shape = <n>,          # <n> = 1: hollow circle, no fill; <n> = 21: circle that can be filled
#             fill = <color 1>,     # color of point fill (optional)
#             color = <color 2>,    # color of point line (optional)
#             size = <s>)          # size of point line (optional)
# <scatterplot> <- <scatterplot> + xlab("<x label>")          # label/caption on x-axis
# <scatterplot> <- <scatterplot> + ylab("<y label>")          # label/caption on x-axis
# <scatterplot> <- <scatterplot> + ggtitle("<scatterplot title>") # scatterplot title

```

```

# Boxplots:
# boxplot(<dataset>$<column name>, xlab = "<column name>") # basic boxplot for <column name>
# boxplot.stats(<dataset>$<column name>)                 # returns the stats used for drawing a boxplot
# ggplot(<dataset>,
#       aes(x = "", y = <column name>, fill = "<color>")) + # ggplot2 boxplots
#       geom_boxplot(width = 0.5) +                       # show boxplot of <column name>
#       stat_boxplot(geom = 'errorbar', width = 0.15) +   # boxplot width
#       guides(fill = FALSE) +                            # show whiskers, control their width
#       xlab("")                                          # no legend (it makes no sense here)
#                                                         # no x-axis label (it makes no sense here)

```

```

# Histograms:
# ggplot(data = <dataset>, mapping = aes(x = <column name>)) +
#   geom_histogram(bins = <nbins>,
#                 fill = "<fill color>",
#                 color = "<line color>")

```

```

# Density graphs:
# ggplot(data = <dataset>,
#       mapping = aes(x = <num. var.>, fill = <fill var.>)) +
#   geom_density(alpha = <value>) +                       # alpha: plot transparency (0-1, optional)
#   theme_bw()

```

```

#####
# ML #
#####

```

```

# Model building and examination:
# <model> <- lm(<y> ~ <x1> + <x2> + ...,                # build/fit the model over the <dataset>;
#            data = <dataset>)                       # <x> and <y> are numeric variables from <dataset>
# <model>                                             # show the model
# coef(<model>)                                     # show the coefficients of the linear model (intercept and slope)
# confint(<model>)                                  # show the confidence intervals for the estimated intercept and slope
# summary(<model>)                                  # show the model statistics
# library(rpart)
# <model> <- rpart(<output variable> ~                # build the tree
#               <predictor variable 1> + <predictor variable 2> + ..., # . to include all variables
#               data = <train dataset>,
#               method = "class",                    # build classification tree

```

```

# control = rpart.control(minsplit = <n>, cp = <q>)) # decrease both for larger tree
# Alternatively:
# <model> <- rpart(<output variable> ~ ., # use almost all vars,
# data = subset(<train dataset>,
# select =
# -c(<predictor variable i> +
# <predictor variable j> + ...)), # excluding some specific ones
# method = "class")
# Alternatively:
# <model> <- rpart(<output variable> ~ ., # use almost all vars,
# data = within(<train dataset>,
# rm(<predictor variable i>, # excluding some specific ones
# <predictor variable j>, ...)
# method = "class")
# library(rattle)
# library(rpart.plot)
# library(RColorBrewer)
# fancyRpartPlot(<decision tree>)
# <model> <- kmeans(x = <normalized dataframe>,
# centers = <k>, # K = <k>
# iter.max = <i>, # max number of iterations allowed, e.g. 20
# nstart = <n>) # no. of initial configurations, e.g. 1000 (report on the best one)
# library(e1071)
# library(caret)
# <folds> = trainControl(method = "cv", number = <k>) # define <k>-fold cross-validation parameters
# <cpGrid> = expand.grid(cp = # specify the range of the cp values to examine
# seq(from = <start value>, to = <end value>, by = <step>))
# train(<output variable> ~ # find the optimal value for cp
# <predictor variable 1> + <predictor variable 2> + ..., # . to include all variables
# data = <train dataset>,
# method = "rpart", # use rpart() to build multiple classification trees
# control = rpart.control(minsplit = <min split>), # default <min split> is 20
# trControl = <folds>, tuneGrid = <cpGrid>) # <folds> and <cpGrid> from above
# <pruned model> <- prune(<model>, cp = <optimal cp value>)
# library(class)
# <model> <- knn(train = <training dataset>, # training data without the output (class) variable
# test = <test dataset>, # test data without the output (class) variable
# cl = <class values for training>, # output (class) variable is specified here
# k = <n>) # <n>: random guess, or obtained from cross-validation
# library(e1071)
# ?naiveBayes
# <model> <- naiveBayes(<output variable> ~ ., # include all predictors from the training set
# data = <training dataset>)
# <model> <- naiveBayes(<output variable> ~
# <var 1> + <var 2> + ..., # include only selected predictors from the training set
# data = <training dataset>)
# Multicollinearity:
# library(car)
# vif(<model>)
# sqrt(vif(<model>)) # variables with sqrt(vif) > 2 (2.5 - disagreement) are problematic

# Making predictions:
# <predictions> <- predict(<model>,
# <test dataframe>,
# interval = "confidence" | # include the confidence interval for the predictions (optional; used only in linear regression)
# "predict") # include prediction intervals (optional)
# <predictions> <- predict(object = <decision tree>,
# newdata = <test dataset>,
# type = "class") # for classification task

```

```

# <predictions> <- predict(object = <NB model>,
#                           newdata = <test dataset>,
#                           type = "raw") # compute probabilities, not classes
# <predictions>[<i1>:<ik>] # examine some of the predictions
# <predictions dataframe> <-
#   data.frame(<observation ID> = <test dataset>$<observation ID column>,
#             <another relevant feature> = <test dataset>$<another relevant feature column>,
#             ...,
#             <output feature> = <test dataset>$<output variable>,
#             <predictions feature> = <predictions>)

# Diagnostic plots:
# par(mfrow = c(2,2)) # set up the plotting panel for 4 graphs
# plot(<model>) # plot the 4 graphs
# par(mfrow = c(1,1)) # reset the plotting panel

# Leverage points:
# plot(<model>, 4, id.n = <k>) # Cook's distance for points in the <model>,
# # highlighting top id.n most extreme values (id.n default: 3)
# <leverage statistic> <- hatvalues(<model>) # <leverage statistic>: high-leverage points in the model
# plot(<leverage statistic>)
# <cutoff leverage> <- 2 * (p + 1) / n # n - no. of observations, p - no. of predictors

# R-squared and RMSE:
# Compute R-squared on the test data for a model:
# R-squared = 1 - RSS/TSS, where RSS is the residual sum of squares, and TSS is the total sum of squares
# <predictions RSS> <-
#   sum((<predictions> - <test dataset>$<output variable>)^2)
# <predictions TSS> <-
#   sum((mean(<train dataset>$<output variable>) - <test dataset>$<output variable>)^2)
# <R-squared> <- 1 - <predictions RSS> / <predictions TSS>
# <R-squared>
# Compute Root Mean Squared Error (RMSE) for a model on the test data
# to see how much error we are making with the predictions:
# RMSE = sqrt(RSS/n)
# <predictions RMSE> <- sqrt(<predictions RSS> / nrow(<test dataset>))
# <predictions RMSE>

# ROC curve (Receiver Operating Characteristic)
# library(pROC)
# <ROC curve parameters> <- # compute ROC curve parameters
#   roc(response = <test dataset>$<output variable>,
#        predictor = <predicted probabilities>[, <1 | 2>]) # col. no. of the "positive class" (can be the No class!)
# <ROC curve parameters>$auc # extract and show AUC
# plot.roc(<ROC curve parameters>, # computed in the previous step
#         print.thres = TRUE, # show the probability threshold (cut-off point) on the plot
#         print.thres.best.method =
#           "youden" | # maximize the sum of sensitivity and specificity (the distance to the diag. line)
#           "closest.topleft") # minimize the distance to the top-left point of the plot
# <ROC coords> <- coords(<ROC curve parameters>, # computed in the previous step
#                       ret = c("accuracy", "spec", "sens", "thr", ...), # ROC curve parameters to return
#                       x = # the coordinates to look for:
#                         "local maximas" | # local maximas of the ROC curve
#                         "best" | ...) # the point with the best sum of sensitivity and specificity, i.e.
# # the same as the one shown on the ROC curve

# Compare multiple clustering results/schemes:
# install.packages("fpc")
# library(fpc)

```

```
# ?cluster.stats
# <comparison criteria> <-
#   c("<criteria 1>",
#     "<criteria 2>", ...)
# <distance matrix> <-
#   dist(x = <normalized dataset>)
# <comparison> <- sapply(list(<clustering 1 name> = <clustering 1>, # <clustering 1> computed by kmeans()
#                           <clustering 1 name> = <clustering 2>, # <clustering 2> computed by kmeans()
#                           FUN = function(x)
#                             cluster.stats(<distance matrix>, x))[<comparison criteria>, ]
# install.packages("knitr")
# library(knitr)
# kable(x = comparison, format = "rst")
```