

Basic statistics with R

Data preparation

Missing values (NA)

<code>is.na(x)</code>	TRUE for each missing value
<code>anyNA(x)</code>	Is there at least one missing value?
<code>sum(is.na(x))</code>	Number of missing values
<code>is.finite(x)</code>	FALSE for NA, NaN and Inf
<code>complete.cases(df)</code>	TRUE for each row without missing values
<code>na.omit(df)</code>	Remove rows with missing values
<code>na.rm = TRUE</code>	Argument of most summary functions to ignore NAs, e.g. <code>mean(x, na.rm = TRUE)</code>
<code>cor(x, y, use = "complete.obs")</code>	Use only complete observations in <code>cor()</code> and <code>cov()</code>

Model functions such as `lm()` remove incomplete rows by default (`na.action = na.omit`).

Random numbers and sampling

<code>set.seed(n)</code>	Set the seed of the random number generator to get reproducible results
<code>sample(x, size)</code>	Random sample without replacement; <code>sample(x)</code> returns a random permutation
<code>sample(x, size, replace = TRUE)</code>	Random sample with replacement (e.g. bootstrap)
<code>replicate(n, expr)</code>	Repeat an expression n times, e.g. for simulations
<code>rnorm(n, mean, sd)</code>	Random numbers from a distribution (all distributions: see <i>Probability distributions</i>)

Transformations

Mathematical functions

<code>round(x, n)</code>	Rounds the elements of x to n decimals (.5 is rounded to the even digit)
<code>signif(x, n)</code>	Rounds the elements of x to n significant digits
<code>ceiling(x)</code>	Rounds to the smallest integers not less than the corresponding elements of x
<code>floor(x)</code>	Rounds to the largest integers not greater than the corresponding elements of x
<code>trunc(x)</code>	Rounds by truncating the values in x toward 0
<code>abs(x)</code>	Computes the absolute values of x
<code>sqrt(x)</code>	Computes the (principal) square root of x
<code>x^(1/3)</code>	Computes the cubic root of x
<code>log(x, base)</code>	Computes logarithms with base $base$ (default is natural log)
<code>log2(x)</code>	Computes binary (i.e., base 2) logarithm
<code>log10(x)</code>	Computes the base 10 logarithm
<code>log1p(x)</code>	Computes $\log(1+x)$
<code>exp(x)</code>	Computes the exponential function
<code>expm1(x)</code>	Computes $\exp(x) - 1$
<code>car::basicPower()</code>	Computes a simple power transformation
<code>car::bcPower()</code>	Transforms the elements in x using the Box-Cox family of scaled power transformations
<code>sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(x)</code>	Trigonometric functions; they compute the sine, cosine, tangent, arc-sine, arc-cosine, arc-tangent, and the two-argument arc-tangent.
<code>scale(x)</code>	Centers and/or scales a vector or matrix x ; to center only use <code>scale=FALSE</code> , to scale only use <code>center=FALSE</code> ; default is <code>center=TRUE</code> , <code>scale=TRUE</code> for <i>z-transformation</i>

Choosing a transformation

Data	Transformation
Right-skewed, positive values	<code>log(x)</code> ; <code>log1p(x)</code> if zeros occur
Counts	<code>sqrt(x)</code>
Proportions (0-1)	<code>asin(sqrt(p))</code> (arcsine) or <code>qlogis(p)</code> (logit)
Unknown power	<code>MASS::boxcox(model)</code> to find λ for <code>car::bcPower()</code>
Different units or scales	<code>scale(x)</code> (z-transformation)
Values to the range 0-1	<code>(x-min(x))/diff(range(x))</code>

Back-transformation: `exp()` for log, `x^2` for sqrt, `sin(y)^2` for arcsine, `plogis()` for logit. Check the model assumptions again after transforming.

Matrix algebra

Matrix functions

<code>t(x)</code>	Transpose x
<code>diag(x)</code>	Construct a diagonal matrix (or extract the diagonal of a matrix)
<code>a %*% b</code>	Matrix multiplication
<code>crossprod(x)</code>	Cross-product $t(x) \%*\% x$ (faster)
<code>det(a)</code>	Determinant of a
<code>solve(a)</code>	Return the matrix inverse of a
<code>solve(a,b)</code>	Solve $a \%*\% x = b$ for x
<code>eigen(a)</code>	Eigenvalues and eigenvectors of a
<code>outer(x, y)</code>	Outer product of two vectors

Example: least-squares estimates of a linear model with design matrix X and response y :
`solve(crossprod(X), crossprod(X, y))`

Descriptive statistics

Functions that return a single value

<code>sum(x)</code>	Sum of the elements of x
<code>prod(x)</code>	Product of the elements of x
<code>mean(x)</code>	Arithmetic mean of the elements of x; <code>trim = 0.1</code> gives a trimmed mean
<code>weighted.mean(x, w)</code>	Mean of x with weights w
<code>median(x)</code>	Median of the elements of x
<code>var(x)</code>	Variance of the elements of x (calculated on $n - 1$)
<code>sd(x)</code>	Standard deviation of x
<code>sd(x)/sqrt(length(x))</code>	Calculation of the standard error of the mean
<code>sd(x)/mean(x)</code>	Coefficient of variation (relative spread)
<code>min(x)</code>	Minimum of the elements of x
<code>max(x)</code>	Maximum of the elements of x
<code>length(x)</code>	Number of elements of x
<code>sum(!is.na(x))</code>	Number of non-missing values
<code>names(which.max(table(x)))</code>	Mode (most frequent value)

Add `na.rm = TRUE` to ignore missing values.

Functions that return single values or matrices

<code>summary(x)</code>	For a numeric vector: minimum, quartiles, median, mean and maximum (plus the number of NAs); for a matrix or data frame these statistics are computed for each column
<code>cor(x, y)</code>	Correlation between x and y, or correlation matrix if they are matrices or data frames; argument <i>method</i> selects Pearson's product moment correlation coefficient (default), Kendall's tau or Spearman's rho
<code>cov(x, y)</code>	Covariance between x and y

Functions that return a vector

<code>range(x)</code>	Returns minimum and maximum of elements of x
<code>pmin(x,y), pmax(x,y)</code>	Returns parallel (element-wise) minima/maxima across several vectors
<code>quantile(x, probs)</code>	Sample quantiles corresponding to the given probabilities, e.g. <code>probs = c(0.25, 0.75)</code>
<code>fivenum(x)</code>	Tukey's five-number summary: minimum, lower hinge, median, upper hinge, maximum
<code>diff(x)</code>	Lagged and iterated differences of vector x
<code>rank(x)</code>	Ranks of the elements of x
<code>IQR(x)</code>	Interquartile range of the x values
<code>mad(x)</code>	Median absolute deviation
<code>cumsum(x)</code>	Cumulative sums of the elements of x
<code>cumprod(x)</code>	Cumulative product of the elements of x
<code>cummin(x)</code>	Cumulative minima of the elements of x
<code>cummax(x)</code>	Cumulative maxima of the elements of x
<code>colSums(x)</code>	Sum of each column
<code>rowSums(x)</code>	Sum of each row
<code>colMeans(x)</code>	Arithmetic mean of each column
<code>rowMeans(x)</code>	Arithmetic mean of each row

Frequencies and tables

<code>table(x)</code>	Frequency table: counts of each value or level of x
<code>table(x, y)</code>	Cross table (contingency table) of two variables
<code>prop.table(t)</code>	Relative frequencies; <code>margin = 1</code> gives proportions per row, <code>margin = 2</code> per column
<code>addmargins(t)</code>	Add row and column sums to a table
<code>xtabs(~ a + b, data)</code>	Cross table from a formula and a data frame
<code>cumsum(table(x))</code>	Cumulative frequencies
<code>cut(x, breaks)</code>	Divide a numeric variable into classes (a factor), e.g. before using <code>table()</code>

Grouped statistics

<code>tapply(x, g, mean)</code>	Apply a function (here the mean) to x separately for each group in g
<code>aggregate(y ~ g, data, mean)</code>	Same with a formula; returns a data frame (several groups: <code>y ~ g1 + g2</code> , all other columns: <code>. ~ g</code>)
<code>split(x, g)</code>	Split x into a list with one element per group, e.g. <code>sapply(split(x, g), sd)</code>
<code>table(g)</code>	Number of observations per group

Shape of a distribution and outliers

Right-skewed	Long tail to the right; mean > median
Left-skewed	Long tail to the left; mean < median
Symmetric	Mean $\approx$ median
<code>e1071::skewness(x)</code>	Skewness coefficient (> 0: right-skewed, < 0: left-skewed)
<code>e1071::kurtosis(x)</code>	Kurtosis (peakedness and tail weight)

Rule of thumb for outliers (boxplot): values below $Q1 - 1.5 \cdot IQR$ or above $Q3 + 1.5 \cdot IQR$.

Graphical summaries in base R

<code>hist(x)</code>	Histogram (number of classes via <i>breaks</i>)
<code>plot(density(x))</code>	Kernel density estimate (smoothed histogram)
<code>boxplot(y ~ g, data)</code>	Box-and-whisker plot per group
<code>stripchart(y ~ g, data)</code>	Plot of the individual values per group
<code>barplot(table(x))</code>	Bar plot of frequencies
<code>plot(x, y)</code>	Scatter plot of two variables
<code>pairs(df)</code>	Scatter plot matrix of all pairs of columns
<code>par(mfrow=c(1,2))</code>	Arrange several plots (rows, columns) in one window
<code>boxplot.stats(x)\$out</code>	Values outside the whiskers (potential outliers)

Inferential statistics

Key concepts of hypothesis testing:

- **H₀**: null hypothesis (no effect, no difference); **H₁**: alternative hypothesis
- **p-value**: probability of the observed result (or a more extreme one) if H₀ is true; H₀ is rejected if $p < \alpha$ (usually 0.05)
- **Type I error** (α): rejecting a true H₀; **type II error** (β): not rejecting a false H₀; **power** = $1 - \beta$
- **Standard error**: standard deviation of an estimate, e.g. of the mean: $sd/\sqrt{n}$
- **Confidence interval**: range of plausible values for a parameter (95 % of such intervals contain the true value in repeated sampling)
- **Parametric** tests assume e.g. normally distributed data (or residuals) and equal variances; **non-parametric** tests use ranks
- Argument **alternative** sets a two-sided ("**two.sided**") or one-sided ("**less**", "**greater**") test

Probability distributions

Gaussian (normal) distribution:

<code>rnorm()</code>	Generates random numbers (rxxx)
<code>pnorm(q)</code>	Returns the probabilities for a given set of quantiles (pxxx)
<code>qnorm(p)</code>	Returns the quantiles for a given set of probabilities (qxxx)
<code>dnorm(x)</code>	Gives the density function (dxxx)

Other standard probability distributions:

(see also ?Distributions; replace **d** by **r**, **p** or **q** as for the normal distribution)

<code>dpois(x, lambda)</code>	Poisson
<code>dnbinom(x, size, prob)</code>	Negative binomial
<code>dbinom(x, size, prob)</code>	Binomial
<code>dmultinom(x, size, prob)</code>	Multinomial
<code>dunif(x, min=0, max=1)</code>	Uniform
<code>dexp(x, rate=1)</code>	Exponential
<code>dlnorm(x, meanlog=0, sdlog=1)</code>	Lognormal
<code>dgamma(x, shape, scale=1)</code>	Gamma
<code>dbeta(x, shape1, shape2)</code>	Beta
<code>dt(x, df)</code>	Student t distribution
<code>df(x, df1, df2)</code>	F-distribution
<code>dchisq(x, df)</code>	(Non-central) chi-squared distribution
<code>dweibull(x, shape, scale=1)</code>	Weibull
<code>dcauchy(x, location=0, scale=1)</code>	Cauchy
<code>dgeom(x, prob)</code>	Geometric

Tests for categorical data and proportions

<code>chisq.test(table)</code>	Chi-squared test of independence for a contingency table (expected counts should be ≥ 5)
<code>chisq.test(x, p=p)</code>	Chi-squared goodness-of-fit test against given probabilities
<code>fisher.test(table)</code>	Fisher's exact test (small counts)
<code>prop.test(x, n)</code>	Test and confidence interval for one or more proportions
<code>binom.test(x, n, p)</code>	Exact binomial test of a proportion

Assumption tests for numeric data

<code>shapiro.test(x)</code>	Performs the Shapiro-Wilk test of normality
<code>ks.test(x,y)</code>	Performs a one- or two-sample Kolmogorov-Smirnov test
<code>var.test(x,y)</code>	Performs an F test to compare the variances of two samples from normal populations
<code>bartlett.test()</code>	Performs Bartlett's test for homogeneity of variance across groups
<code>fligner.test()</code>	Performs the Fligner-Killeen test for homogeneity of variance (robust against non-normality)
<code>car::leveneTest(y, group)</code>	Computes Levene's test for homogeneity of variance across groups
<code>car::durbinWatsonTest()</code>	Performs Durbin-Watson test for autocorrelated errors

Tests react to sample size: with large samples tiny deviations become significant, with small samples they are missed. Always check graphically as well (`hist()`, `qqnorm()`, `boxplot()`).

Simple tests for numeric data

1- and 2-sample tests

<code>t.test(x)</code>	1-sample t-test; mu sets the value to compare with, conf.level the confidence level
<code>t.test(x,y)</code>	2-sample independent t-test (Welch test by default; use var.equal=TRUE for Student's t-test)
<code>t.test(x,y, paired=TRUE)</code>	2-sample dependent t-test
<code>t.test(y ~ g, data)</code>	2-sample t-test with formula and grouping variable
<code>t.test(x)\$conf.int</code>	95% confidence interval of the mean
<code>wilcox.test(x)</code>	Non-parametric 1-sample Wilcoxon signed rank test
<code>wilcox.test(x,y)</code>	Non-parametric 2-sample Mann-Whitney U test (or Wilcoxon rank sum test)
<code>wilcox.test(x,y, paired=TRUE)</code>	Non-parametric 2-sample Wilcoxon signed rank test
Multiple sample tests	
<code>kruskal.test(x, g)</code>	Performs the non-parametric Kruskal-Wallis rank sum test
<code>friedman.test(y, groups)</code>	Performs a Friedman rank sum test with unreplicated blocked data
Correlation tests	
<code>cor.test(x, y)</code>	Test for correlation between paired samples; method = " spearman " or " kendall " for rank correlations

Multiple testing and post-hoc tests

<code>p.adjust(p, method="holm")</code>	Adjust p-values for multiple comparisons; further methods: " bonferroni ", " BH " (false discovery rate)
<code>pairwise.t.test(x, g)</code>	Pairwise t-tests between all groups (p-values adjusted)
<code>pairwise.wilcox.test(x, g)</code>	Pairwise Wilcoxon tests between all groups

Power analysis

Compute the power or determine parameters (e.g. sample size) to obtain a target power; the parameter to be calculated is left as NULL: Base R (package *stats*)

<code>power.t.test()</code>	One- or two- sample t test
<code>power.anova.test()</code>	Balanced one-way ANOVA
<code>power.prop.test()</code>	Two-sample test for proportions

Further packages

<code>pwr::pwr.t.test()</code>	t tests (1-sample, 2-sample independent, paired)
<code>pwr::pwr.t2n.test()</code>	t test (2-sample, unequal <i>n</i>)
<code>MKpwr::</code>	Welch t test
<code>power.welch.t.test()</code>	
<code>WMWssp::WMWssp()</code>	Mann-Whitney <i>U</i> test (only sample size <i>n</i> ; needs data of a reference group)
<code>pwr::pwr.chisq.test()</code>	Chi-squared test
<code>pwr::pwr.anova.test()</code>	Balanced one-way ANOVA (equal <i>n</i>)
<code>MultNonParam::kwpower()</code>	Kruskal-Wallis test
<code>MultNonParam::kwsamplesize()</code>	
<code>pwr::pwr.r.test()</code>	Correlation
<code>WebPower::</code>	Linear regression
<code>wp.regression()</code>	

Linear models

One framework: the linear model

Explanatory variables	Name and formula
one continuous	Simple linear regression: $y \sim x$
several continuous	Multiple regression: $y \sim x_1 + x_2$
one or more categorical (factors)	ANOVA (one- or multi-way): $y \sim f_1 + f_2$
continuous and categorical	ANCOVA: $y \sim x + f$ (parallel lines), $y \sim x * f$ (different slopes)

All of these are linear models. `lm()` and `aov()` fit the same model and differ mainly in the default output: `summary(model)` shows the coefficients (for factors: differences to the reference level), `anova(model)` the variance table per term.

Fitting and results

<code>lm(y ~ x, data)</code>	Fits a linear model; response <i>Y</i> has to be continuous, explanatory variables <i>X</i> can be continuous or categorical (function can be used for linear regressions, ANCOVAs and ANOVAs)
<code>aov(y ~ x, data)</code>	Fits the same model as <code>lm()</code> (by a call to <code>lm</code> for each stratum), but <code>summary()</code> prints an ANOVA table; needed for <code>TukeyHSD()</code> and <code>Error()</code> strata
<code>coef(model)</code>	Returns the estimated coefficients (sometimes with their standard-errors)
<code>summary(model)</code>	For <code>lm</code> objects: table of parameter estimates, standard errors, t-statistics and p values, plus R^2 , adjusted R^2 and the overall F-test; for <code>aov</code> objects: type-I ANOVA table with SS, MS, F-ratio and p-values. Can also be called explicitly via <code>summary.lm()</code> or <code>summary.aov()</code>
<code>confint(model)</code>	Computes confidence intervals for one or more parameters in a fitted model
<code>residuals(model)</code>	Returns the model residuals (short form: <code>resid(model)</code>)
<code>fitted(model)</code>	Returns the fitted values
<code>predict(model)</code>	Returns predicted values for observed or generated predictor values (via argument <i>newdata</i>); <code>interval = "confidence"</code> or "prediction" adds intervals; can also be defined explicitly via <code>predict.lm()</code>

Formula

<code>formula()</code>	Provide a way of extracting formulae which have been included in other objects
<code>formula(y ~ x)</code>	<i>y</i> as a function of the fixed (continuous or categorical) <i>x</i>
<code>formula(y ~ Error(x))</code>	Error strata in <code>aov()</code> , e.g. $y \sim A + \text{Error}(\text{Subject})$
<code>formula(y ~ x1 + x2)</code>	<i>y</i> as a function of <i>x1</i> and <i>x2</i>
<code>formula(y ~ x1 + x2 + x1:x2)</code>	<i>y</i> as a function of <i>x1</i> and <i>x2</i> , including their interaction
<code>formula(y ~ x1 * x2)</code>	Shorthand notation for the interaction inclusion
<code>formula(y ~ x1 + x2 in % x1)</code>	<i>y</i> as a function of <i>x1</i> and <i>x2</i> , which is nested in <i>x1</i>
<code>or x1/x2 or x1+x1:x2</code>	<i>y</i> as a function of <i>x1</i> and <i>x2</i> , which is nested in <i>x1</i>
<code>! or x1/x2 or x1+x1:x2</code>	<i>y</i> as a function of <i>x1</i> and <i>x2</i> , which is nested in <i>x1</i>
<code>formula(y ~ .)</code>	<i>y</i> as a function of all other columns of the data
<code>formula(y ~ x - 1)</code>	Model without intercept
<code>formula(y ~ I(x^2))</code>	Arithmetic operations inside a formula must be wrapped in <code>I()</code> ; alternative for polynomials: <code>poly(x, 2)</code>
<code>formula(log(y) ~ x)</code>	Functions (e.g. transformations) can be used directly in the formula
<code>update(model, .~. -x2)</code>	Is used to update model formulae. This typically involves adding or dropping terms

Testing terms and comparing groups

These functions work for regression, ANOVA and ANCOVA models alike:

<code>anova(model)</code>	Computes the sequential (type-I) analysis of variance table of a fitted model (or compares several models); the order of the terms in the formula matters
<code>car::Anova(model)</code>	Computes type-II or type-III ANOVA tables for model objects produced by <i>aov</i> , <i>lm</i> and other model functions; type III requires sum-to-zero contrasts
<code>drop1(fit, ~., test="F")</code>	Computes marginal SS and F tests by dropping each term in turn, respecting marginality (main effects that are part of an interaction are not tested)
<code>oneway.test(y ~ g)</code>	One-way ANOVA that does not assume equal variances (Welch)
<code>TukeyHSD(model)</code>	Computes Tukey Honest Significant Differences for <i>post-hoc</i> comparison (for <i>aov</i> objects)
<code>model.tables(model)</code>	Computes summary tables for model fits, especially complex <i>aov</i> fits
<code>plot.design()</code>	Plots univariate effects of a design or model
<code>interaction.plot()</code>	Two-way interaction plot

Factors and contrasts

By default R uses treatment contrasts: the first level of a factor is the reference and the coefficients are differences to it.

<code>relevel(f, ref)</code>	Set the reference level of a factor
<code>contrasts()</code>	Set and view the contrasts associated with a factor
<code>model.matrix()</code>	Creates a design (model) matrix, e.g. by expanding factors to dummy variables according to the contrasts

Model diagnostics

Assumptions of linear models: linear relationship, independent errors, normally distributed residuals, equal variance of residuals (homoscedasticity), no influential outliers.

<code>plot(model)</code>	Produces diagnostic plots to check model assumptions and influential data points for an <i>lm</i> object: residuals vs. fitted (linearity), Q-Q plot (normality), scale-location (equal variance), residuals vs. leverage
<code>cooks.distance(model)</code>	Computes Cook's distance (influence of each observation); large values (e.g. > 1) indicate influential points
<code>hatvalues(model)</code>	Leverage of each observation
<code>rstandard(model)</code>	Standardized residuals
<code>rstudent(model)</code>	Studentized residuals
<code>qqnorm(y)</code>	Produces a normal quantile-quantile plot of the values in <i>y</i>
<code>qqplot()</code>	Produces quantile-quantile plot
<code>qqline()</code>	Adds a line to a normal quantile-quantile plot (created with <code>qqnorm()</code>), which passes through the first and third quartiles
<code>acf()</code>	Computes (and by default plots) estimates of the autocovariance or autocorrelation function
<code>pacf()</code>	Computes the partial autocorrelation function
<code>ccf()</code>	Computes the cross-correlation or cross-covariance of two univariate series
<code>car::vif(model)</code>	Calculates variance-inflation factors to detect collinearity of predictors (values > 5 indicate problems)

Model selection

<code>AIC(model1, model2)</code>	Computes the Akaike information criterion or AIC; the lower, the better the trade-off between fit and complexity (models must be fitted to the same data)
<code>BIC(model1, model2)</code>	Computes the Bayesian information criterion; penalizes complexity more strongly than the AIC
<code>step(model)</code>	Select a formula-based model by AIC in a stepwise algorithm (backward and forward selection)
<code>anova(m1, m2, test="F")</code>	Compares two (or more) nested <i>lm</i> or <i>aov</i> models based on the F-test
<code>drop1(model, test="F")</code>	Drops each term in turn and compares the fits (see <i>Testing terms</i>)

Tidyverse: the *broom* and *modelr* package

`library(broom)` (installed with, but not attached by, `library(tidyverse)`)

Turns model objects (*lm*, *glm*, *t.test*, ...) into tibbles:

<code>tidy(model)</code>	One row per model term: estimates, std.errors, test statistic and p-value; add confidence intervals with <code>conf.int = TRUE</code>
<code>augment(model)</code>	One row per observation: observational data plus fitted values, residuals, Cook's D, etc.
<code>augment(model, newdata = nd)</code>	Predictions for new data; add <code>interval = "prediction"</code> (or <code>"confidence"</code>) for intervals
<code>glance(model)</code>	One row per model: summary statistics such as R^2 , AIC and BIC

The *modelr* package is superseded (see *tidymodels* for a modern framework); `broom::augment()` covers most use cases. The following functions still work:

```
modelr::add_predictions(data, model)
Adds predictions from a single model to a data frame
modelr::add_residuals(data, model)
Adds residuals from a single model to a data frame
modelr::data_grid(data, x1, x2)
Generates an evenly spaced grid of points from the data
```