Package 'STATassist'

September 5, 2026

Title Standardised Statistical Comparison Workflows

Version 1.0.0

Description Runs the applicable tests for a comparison in one call and returns standardised result tables. One-sample, two-group, multi-group, factorial and categorical workflows report parametric, rank-based and robust results side by side with effect sizes, confidence intervals and multiplicity-adjusted p-values. Supervised fits, embeddings, clustering and simulators follow the same result contracts. Methods include those of Welch (1947) <doi:10.1093/biomet/34.1-2.28>, Wilcoxon (1945) <doi:10.2307/3001968>, Mann and Whitney (1947) <doi:10.1214/aoms/1177730491>, Kruskal and Wallis (1952) <doi:10.1080/01621459.1952.10483441>, Friedman (1937) <doi:10.1080/01621459.1937.10503522>, Tukey (1949) <doi:10.2307/3001913>, Dunn (1964) <doi:10.1080/00401706.1964.10490181>, Yuen (1974) <doi:10.1093/biomet/61.1.165>, Brunner and Munzel (2000) <doi:10.1002/(SICI)1521-4036(200001)42:1%3C17::AID-BIMJ17%3E3.0.CO;2-U>, Algina, Keselman and Penfield (2005) <doi:10.1037/1082-989X.10.3.317>, DeLong, DeLong and Clarke-Pearson (1988) <doi:10.2307/2531595>, Sun and Xu (2014) <doi:10.1109/LSP.2014.2337313>, Pencina, D'Agostino, D'Agostino and Vasan (2008) <doi:10.1002/sim.2929>, and Pencina, D'Agostino and Steyerberg (2011) <doi:10.1002/sim.4085>.

License MIT + file LICENSE

URL https://github.com/hiows/STATassist

BugReports https://github.com/hiows/STATassist/issues

Encoding UTF-8

Depends R (>= 4.1.0)

Imports caret, dbscan, glmnet, grDevices, graphics, kernlab, randomForest, Rtsne, stats, umap, utils

Suggests jsonlite, pROC, testthat (>= 3.0.0), withr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Wonseok Oh [aut, cre] (ORCID: <https://orcid.org/0009-0002-0687-8466>)

Maintainer Wonseok Oh <hiows97@gmail.com>

Repository CRAN

Date/Publication 2026-09-05 12:50:02 UTC

Contents

`as.table.sa_categorical` *The contingency table a categorical comparison was run on*

Description

`$cells` is the canonical form, one row per cell, because that is the shape that survives being written out as JSON with its labels attached. A table is the shape to read it in, so it is built on request rather than stored twice and left to drift.

Usage

```
## S3 method for class 'sa_categorical'
as.table(x, ...)
```

Arguments

`x`	A categorical comparison, as returned by `compare_categorical_groups()`.
`...`	Ignored, present for consistency with `as.table()`.

Details

The rows dropped for a missing value or for a level outside `category_lv` are already gone, so this is the table the tests were run on rather than a fresh `table()` of the input.

Value

A two-dimensional `table()` of counts, its dimnames named after the two axes of the design.

Examples

```
res <- compare_categorical_groups(
  data.frame(smoker = rep(c("y", "n"), each = 30),
             grade  = c(rep(c("high", "low"), c(22, 8)),
                        rep(c("high", "low"), c(9, 21))))
)
as.table(res)

## Which is the table the tests were run on, so it feeds `stats` directly.
stats::chisq.test(as.table(res))
```

`center_by_control` *Centre every feature on the control group*

Description

Divides, or on the log2 scale subtracts, the centre of the control group out of each feature, so that the control lands on 1 (raw) or 0 (log2) and every other observation reads as its distance from the control rather than as a measurement of its own. The `data` that comes back is the `data` that went in with the `feats` columns replaced, which is what lets the same call be handed straight to a comparison or to `draw_heatmap()`.

Usage

```
center_by_control(
  data,
  feats,
  group,
  group_lv,
  control_label = group_lv[1],
  fc_mean = c("arith", "geom"),
  input_scale = c("raw", "log2")
)
```

Arguments

`data`	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
`feats`	Character vector of numeric column names in `data` to centre. Columns not named here are returned untouched.
`group`	Grouping vector with one entry per row of `data`.
`group_lv`	Character vector of at least two group levels. Unlike in a comparison, rows belonging to another level are kept: they are centred on the same baseline and take no part in computing it, which leaves the result the same length as the `group` the comparison still has to be given. The comparison drops them itself.
`control_label`	The level whose centre is divided out. Defaults to `group_lv[1]`, the reference the order already asked for, so naming it is only needed to point the baseline at a level that is not first.
`fc_mean`	Which centre of the control group is removed, `"arith"` for the arithmetic mean or `"geom"` for the geometric mean. The geometric mean requires strictly positive values. Defaults to `"geom"` when `input_scale = "log2"` and to `"arith"` otherwise, exactly as it does in the comparisons.
`input_scale`	The scale `data` arrives on, `"raw"` or `"log2"`. This decides the operation: a ratio on the raw scale, a difference on the log2 scale. The centred data stays on the scale it arrived on, so the same `input_scale` is what the comparison should be given afterwards.

Details

The arguments are the ones `compare_two_groups()` and `compare_multiple_groups()` already take, in the same order and with the same defaults, because the point of the function is that one set of arguments describes both steps:

```
centred <- center_by_control(data, feats, group, group_lv,
                             input_scale = "log2")
compare_two_groups(centred, feats, group, group_lv, input_scale = "log2")
```

`fc_mean` is the comparison's own argument rather than a centring method of this function's, and it is resolved the same way: the geometric mean by default on the log2 scale, where it is the convention, and the arithmetic mean otherwise. Passing the same value to both is what keeps the baseline removed here identical to the centre the comparison divides in its `effect` table.

Two or more levels are accepted, so the same call serves a two-group and a multi-group design. Only the control level takes part in the baseline; the rest are centred on it.

Value

The `data` that was passed in, as a data.frame, with the `feats` columns replaced by their control-relative values. Row count, row order, row names and every column that is not a feature are left as they arrived. A feature whose baseline could not be taken comes back as an `NA` column and is named in a warning.

What the comparison reports afterwards

The transformation is one constant per feature applied to every row of it, so most of what a comparison reports does not move:

`fold_change` **and** `log2fc` Unchanged. Both group centres are divided by the same baseline, so their ratio survives it. This holds for all four combinations of `fc_mean` and `input_scale`.

p-values Unchanged, in every test family. On the log2 scale the baseline is subtracted, and the tests are shift invariant; on the raw scale it is divided, and dividing by a positive constant leaves the t statistic and every rank untouched.

The reference centre Becomes 1. It is `y_center` in `compare_two_groups()` and `ref_center` in `compare_multiple_groups()`, and once it is 1 the other centre in the row is the fold change itself.

`mean_diff`**,** `hl_shift` **and** `trim_diff` Unchanged on the log2 scale. On the raw scale they are divided by the baseline along with the data, since they are differences rather than ratios.

The reference centre lands on 1 because the observations used here and the ones the comparison centres are the same. Under `paired = TRUE` a comparison keeps complete pairs or complete subjects only, so if any are dropped its control centre is taken on fewer rows than the baseline was and comes back near 1 rather than at it. The ratios are unaffected either way.

See Also

`compare_two_groups()` and `compare_multiple_groups()`, whose arguments this shares, and `draw_heatmap()`, which takes the result with `scale = "none"` to show departure from the control rather than from each feature's own mean.

Examples

```
feats <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")
group_lv <- levels(iris$Species)

## setosa is the first level and so the control. Every measurement is now a
## multiple of the setosa mean, and setosa itself centres on 1.
centred <- center_by_control(iris, feats, iris$Species, group_lv)
colMeans(centred[iris$Species == "setosa", feats])

## The comparison takes the centred data with its arguments unchanged, and
## reports the fold changes it reported before the centring.
before <- compare_multiple_groups(iris, feats, iris$Species, group_lv,
                                  posthoc = FALSE, diagnose = FALSE)
after <- compare_multiple_groups(centred, feats, iris$Species, group_lv,
                                 posthoc = FALSE, diagnose = FALSE)
cbind(before = before$effect$log2fc, after = after$effect$log2fc)

## What did change is the reference centre, which is now 1 by construction,
## so `extreme_center` reads as the fold change on its own.
after$effect[c("ref_center", "extreme_center", "fold_change")]

## On the log2 scale the baseline is subtracted instead, and the control
## centres on 0. Rounded, because the subtraction leaves floating point dust
```

```
## rather than a clean zero.
d <- iris[iris$Species != "setosa", ]
lg <- center_by_control(log2(d[feats]), feats, d$Species,
                        c("versicolor", "virginica"),
                        input_scale = "log2")
round(colMeans(lg[d$Species == "versicolor", ]), 12)

## Pointing the baseline at the other level is one argument rather than a
## rewritten `group_lv`.
virg <- center_by_control(iris, feats, iris$Species, group_lv,
                          control_label = "virginica")
colMeans(virg[iris$Species == "virginica", feats])
```

`cluster_dbscan` *Cluster by finding the dense regions*

Description

Grows a cluster out of every point that has at least `min_pts` neighbours within `eps` of it, joining the neighbourhoods that overlap, and leaves the points that never fell into one as noise. How many clusters there are is the answer rather than the question, and it can be none.

Usage

```
cluster_dbscan(
  data,
  feats = NULL,
  cluster_scale = c("samples", "features"),
  center = TRUE,
  scale = TRUE,
  eps = NULL,
  min_pts = NULL
)
```

Arguments

`data`	A data.frame or a matrix in wide format, one row per sample and one column per feature.
`feats`	Column names to cluster on, or `NULL` for every numeric column of `data`.
`cluster_scale`	Which margin becomes the points being clustered: `"samples"`, the default, or `"features"`. `design$point_type` reports it.
`center`, `scale`	Whether to centre each feature and divide it by its standard deviation first. Both always apply to the **columns of** `data`, whatever `cluster_scale` is. Scaling matters here because `eps` is one radius applied to all of them at once.
`eps`	The radius of a neighbourhood, in the units of the matrix being clustered, or `NULL` to derive it as the radius reaching the `min_pts - 1`th neighbour of 95% of the points. See the details.

`min_pts` How many points must be within `eps` of a point, itself included, before it can be the core of a cluster. `NULL` derives it from the number of variables, capped at half the points; the derived value is reported in a message.

Details

The input is the wide format the comparison functions take: **one row per sample and one column per feature**. Which margin of it becomes the thing being clustered is `cluster_scale`, and `design$point_type` reports the answer.

Value

An object of class `sa_cluster`, a plain list. The slots are `cluster_hclust()`'s, with `analysis` `"dbscan"`, `fit` the `dbscan::dbscan()` object, and `design$n_clusters` and `design$n_noise` both answers rather than arguments.

Noise is a result, not a failure

A point that never joined a neighbourhood gets cluster `0`, which is `dbscan::dbscan()`'s convention and this contract's. It has no silhouette, since noise is not a cluster to be near or far from, and `clusters` has no row for it; `design$n_noise` is the count. All points being noise is a possible answer and means the density asked for is not present, which is usually `eps` being too small or `min_pts` too large for the data.

Choosing eps

`eps` is a radius in the units of the matrix being clustered, which is why there is no default constant that could be right. Left as `NULL` it is derived from the distances actually present: every point's distance to its `min_pts - 1`th neighbour is collected, and `eps` is the 95th percentile of those. The rule therefore reads as "assume about one point in twenty is noise, and set the radius that leaves that many outside", which is a statement worth disagreeing with rather than a number off a curve.

The quantity being read is the one `dbscan::kNNdistplot()` plots and the manual procedure reads a knee off by eye. The knee is not what is taken, because locating it arithmetically fails on a curve that rises gradually rather than turning; the probe in `Test/cursor_test/2026_08_15/` has the case. Either way it is a heuristic: `parameters$eps_source` records whether the value was supplied or derived, and a derived one is said out loud when it is used.

Note that this `eps` is a distance. `cluster_snn()` has an argument of the same name that counts shared neighbours, because the algorithm it comes from named it that; the two are not comparable.

Which margin is clustered

`cluster_scale = "samples"`, the default, puts one point per row of `data`, and `"features"` puts one point per column. See `cluster_hclust()`, which documents the same argument and why transposing `data` by hand is a different analysis.

See Also

`cluster_snn()`, the other density method here, which measures closeness by how many neighbours two points share rather than by a radius, and `cluster_kmeans()`, which is told the number of groups and places every point.

Examples

```
## Two well-separated blobs and a couple of strays. The count is not an
## argument, so finding two is the result rather than the setup.
set.seed(1)
blobs <- rbind(
  matrix(rnorm(60, mean = -3), ncol = 2),
  matrix(rnorm(60, mean =  3), ncol = 2),
  cbind(c(-12, 12), c(12, -12))
)
colnames(blobs) <- c("x", "y")
res <- cluster_dbscan(blobs, scale = FALSE)
res
table(res$assignments$cluster)

plot(blobs, col = res$assignments$cluster + 1L, pch = 16,
     main = "cluster 0, in black, is noise")

## On a planted two-group structure, with the radius left to be derived.
sim <- simulate_two_groups(n_feats = 30, n_up = 5, n_down = 5, seed = 3)
by_samp <- cluster_dbscan(sim$args$data)
by_samp$parameters$eps_source
table(cluster = by_samp$assignments$cluster, group = sim$args$group)
```

---

cluster_hclust *Cluster by building a tree and cutting it*

Description

Merges the two closest points, then the two closest groups, and keeps going until everything is one group; then cuts the resulting tree so that `n_clust` groups fall out. What comes back is one cluster label per point, beside the `stats::hclust()` tree the labels were cut from.

Usage

```
cluster_hclust(
  data,
  feats = NULL,
  cluster_scale = c("samples", "features"),
  center = TRUE,
  scale = TRUE,
  n_clust = 2,
  dist_method = c("euclidean", "manhattan", "correlation"),
  hclust_method = c("average", "complete", "ward.D2")
)
```

Arguments

data	A data.frame or a matrix in wide format, one row per sample and one column per feature. This is the same layout compare_two_groups() and draw_heatmap() take. Row names are kept as the sample labels, repeated ones included; rows without a name are labelled by position.
feats	Column names to cluster on, or NULL for every numeric column of data. A non-numeric column is left out with a message, so a frame that carries a grouping column alongside the measurements can be passed as it is.
cluster_scale	Which margin becomes the points being clustered: "samples", the default, for one point per row of data, or "features" for one point per column. design$point_type reports it. See the details: this is not the same as transposing data yourself.
center, scale	Whether to centre each feature and divide it by its standard deviation before measuring anything. Scaling is on by default because features are not measured on a common scale, and without it the feature with the widest units decides which points are close. Both always apply to the **columns of** data, whatever cluster_scale is.
n_clust	How many groups to cut the tree into. The tree itself does not depend on this, so $fit can be cut again at another k.
dist_method	What "close" means. "correlation" is 1 - cor(), which compares the shape of a profile rather than its level; the other two are handed to stats::dist().
hclust_method	Linkage handed to stats::hclust(), which is what "close" means for two groups rather than two points.

Details

The input is the wide format the comparison functions take: **one row per sample and one column per feature**. Which margin of it becomes the thing being clustered is cluster_scale, and design$point_type reports the answer.

Value

An object of class sa_cluster, a plain list.

analysis "hclust".

points Labels of the things that were clustered — samples or features, as cluster_scale asked — in the row order assignments follows.

design What was clustered: point_type, the feats kept and any dropped_feats, the counts n_samples, n_used, n_dropped and n_feats, and the two counts of the answer, n_clusters and n_noise. n_noise is always 0 here: a tree places every point.

parameters The choices as they were used.

assignments One row per point: points, cluster and silhouette.

clusters One row per cluster: cluster, size and the mean silhouette of its members.

engine What computed the clustering.

fit The stats::hclust() object. This is the slot that is not portable; dropping it leaves an object that writes out as JSON.

metadata Package version, R version, platform and timestamp.

Which margin is clustered

`cluster_scale = "samples"`, the default, puts one point per row of `data`, and the question is which samples resemble each other. `"features"` puts one point per column, and the question is which features move together. The second is the one this and `draw_heatmap()` agree on: the features are standardised first and the transpose is clustered as it stands, which is also what `perform_pca(embedding_scale = "features")` reports on.

Transposing `data` by hand instead is a third analysis. `scale()` standardises **columns**, so on the transpose it standardises samples, and the picture looks right while answering a different question. `cluster_scale` exists so that the transpose never has to be taken by hand.

Choosing the distance and the linkage

`dist_method` decides what "close" means and `hclust_method` decides what "close" means for two groups rather than two points. `"correlation"` is `1 - cor()`, so two points a constant apart are at distance zero: use it when the shape of a profile is the question and its level is not. The default `"average"` linkage is the one that neither chains clusters into strings the way single linkage does nor insists they come out round the way `"ward.D2"` does.

The tree is built on exactly the distance `draw_heatmap()` builds its dendrogram from, so a heatmap drawn with the same `dist_method` and `hclust_method` is showing this clustering and not a near relative of it.

What is dropped before anything runs

Rows that are not complete and finite across `feats` go before the distance is measured, and `design$n_dropped` reports how many. This is the listwise deletion the rest of the package uses; nothing is imputed. A feature that takes a single value cannot be scaled, so with `scale = TRUE` it is left out with a message and named in `design$dropped_feats`.

See Also

`cluster_kmeans()`, which is told the same `n_clust` and answers without a tree, and `cluster_dbscan()`, which is not told it at all. `draw_heatmap()` draws the tree this cuts.

Examples

```
## Three species, and the clustering was not told there were three.
res <- cluster_hclust(iris[1:4], n_clust = 3)
res
table(cluster = res$assignments$cluster, species = iris$Species)

## The tree is `$fit`, so another cut costs nothing.
table(stats::cutree(res$fit, k = 2))

## On a planted two-group structure, clustering the samples.
sim <- simulate_two_groups(n_feats = 30, n_up = 5, n_down = 5, seed = 3)
by_samp <- cluster_hclust(sim$args$data, n_clust = 2)
table(cluster = by_samp$assignments$cluster, group = sim$args$group)

## `cluster_scale = "features"` asks the other question: which features move
```

```
## together. The correlation distance is the one that ignores their levels.
by_feat <- cluster_hclust(sim$args$data, cluster_scale = "features",
                          n_clust = 3, dist_method = "correlation")
by_feat
split(by_feat$points, by_feat$assignments$cluster)
```

`cluster_kmeans` *Cluster by moving centres until they stop*

Description

Places `n_clust` centres, gives every point to its nearest one, moves each centre to the mean of the points it was given, and repeats until nothing moves. What comes back is one cluster label per point, beside the `stats::kmeans()` object holding the centres.

Usage

```
cluster_kmeans(
  data,
  feats = NULL,
  cluster_scale = c("samples", "features"),
  center = TRUE,
  scale = TRUE,
  n_clust = 2,
  n_start = 25,
  iter_max = 100,
  seed = NULL
)
```

Arguments

`data`	A data.frame or a matrix in wide format, one row per sample and one column per feature.
`feats`	Column names to cluster on, or `NULL` for every numeric column of `data`.
`cluster_scale`	Which margin becomes the points being clustered: `"samples"`, the default, or `"features"`. `design$point_type` reports it.
`center, scale`	Whether to centre each feature and divide it by its standard deviation first. Both always apply to the **columns of** `data`, whatever `cluster_scale` is. Scaling matters more here than anywhere else in this family, since a mean is taken in the units the features arrived in.
`n_clust`	How many centres to place. Unlike the two density methods, this is the number of clusters that will come back.
`n_start`	How many random starts to try, keeping the best. The default of 25 overrides `stats::kmeans()`'s own default of 1.
`iter_max`	Most iterations one start may take before it is abandoned.
`seed`	Seed for the starts, or `NULL` to leave the random stream alone. The stream is restored afterwards either way.

Details

The input is the wide format the comparison functions take: **one row per sample and one column per feature**. Which margin of it becomes the thing being clustered is `cluster_scale`, and `design$point_type` reports the answer.

Value

An object of class `sa_cluster`, a plain list. The slots are `cluster_hclust()`'s, with `analysis` `"kmeans"`, `fit` the `stats::kmeans()` object and `design$n_noise` always 0, since every point is given to a centre.

Reproducibility

The starting centres are random, so two calls can disagree. Two defences are on by default and neither replaces the other. `n_start = 25` runs the whole thing twenty-five times from different starts and keeps the one with the smallest within-cluster sum of squares, which makes a bad local optimum unlikely rather than impossible. `seed` makes the run exactly repeatable, and it restores the random stream afterwards, so seeding this call does not quietly reseed whatever the caller does next.

Neither is a guarantee that `n_clust` is the right number. k-means always returns the number of groups it was asked for, and `clusters$silhouette` is where to look for whether they are groups: a cluster near zero is one whose members are about as close to another cluster as to their own.

Which margin is clustered

`cluster_scale = "samples"`, the default, puts one point per row of `data`, and `"features"` puts one point per column. See `cluster_hclust()`, which documents the same argument and why transposing `data` by hand is a different analysis.

What is dropped before anything runs

Rows that are not complete and finite across `feats` go before the centres are placed, and `design$n_dropped` reports how many. A feature that takes a single value cannot be scaled, so with `scale = TRUE` it is left out with a message and named in `design$dropped_feats`.

See Also

`cluster_hclust()`, which is told the same `n_clust` and returns the tree it cut, and `cluster_dbscan()`, which derives the count from the density instead and may refuse to place a point.

Examples

```
## Three species, and the clustering was not told there were three.
res <- cluster_kmeans(iris[1:4], n_clust = 3, seed = 1)
res
table(cluster = res$assignments$cluster, species = iris$Species)

## The centres are on `$fit`, in the scaled units the clustering ran in.
res$fit$centers
```

```
## On a planted two-group structure. The group was never shown to the engine.
sim <- simulate_two_groups(n_feats = 30, n_up = 5, n_down = 5, seed = 3)
by_samp <- cluster_kmeans(sim$args$data, n_clust = 2, seed = 1)
table(cluster = by_samp$assignments$cluster, group = sim$args$group)

## Asking for more groups than there are shows up in the silhouettes rather
## than in an error: k-means returns whatever number it was asked for.
cluster_kmeans(sim$args$data, n_clust = 5, seed = 1)$clusters
```

`cluster_snn` *Cluster by how many neighbours points have in common*

Description

Builds a graph in which every point keeps its `k` nearest neighbours, links two points that share at least `eps` of them, and grows a cluster out of every point with at least `min_pts` such links. Points that never joined one are left as noise. How many clusters there are is the answer rather than the question.

Usage

```
cluster_snn(
  data,
  feats = NULL,
  cluster_scale = c("samples", "features"),
  center = TRUE,
  scale = TRUE,
  k = NULL,
  eps = NULL,
  min_pts = NULL
)
```

Arguments

`data`	A data.frame or a matrix in wide format, one row per sample and one column per feature.
`feats`	Column names to cluster on, or `NULL` for every numeric column of `data`.
`cluster_scale`	Which margin becomes the points being clustered: `"samples"`, the default, or `"features"`. `design$point_type` reports it.
`center, scale`	Whether to centre each feature and divide it by its standard deviation first. Both always apply to the **columns of** `data`, whatever `cluster_scale` is.
`k`	How many nearest neighbours each point keeps. `NULL` derives it as the square root of the number of points and reports what it came out as.
`eps`	How many neighbours two points must **share** before they are linked, a whole number from 1 to `k`. Not a distance; see the details. `NULL` uses `k / 2`.
`min_pts`	How many links a point needs in that graph before it can be the core of a cluster. `NULL` uses `k / 2`.

Details

The input is the wide format the comparison functions take: **one row per sample and one column per feature**. Which margin of it becomes the thing being clustered is `cluster_scale`, and `design$point_type` reports the answer.

Value

An object of class `sa_cluster`, a plain list. The slots are `cluster_hclust()`'s, with `analysis` `"snn"`, `fit` the `dbscan::sNNclust()` object, and `design$n_clusters` and `design$n_noise` both answers rather than arguments.

`eps` here counts neighbours, and in cluster_dbscan() it is a distance

The two functions have an argument of the same name meaning two different things, which is the algorithms' doing rather than this package's: `dbscan::dbscan()` and `dbscan::sNNclust()` both call their threshold `eps`, and renaming one of them here would make this package's documentation disagree with the engine's. In `cluster_dbscan()` `eps` is a radius in the units of the data and any positive number is meaningful. Here it is a count of shared neighbours, so it is a whole number between 1 and `k`, and a value of `k / 2` says "half of what each of you considers close is the same points".

Why this rather than a radius

One radius has to be right everywhere, so DBSCAN cannot find a tight cluster and a loose one in the same call: the `eps` that reaches across the loose one merges the tight one into its surroundings. Shared neighbours are relative, so a point in a sparse region is still close to its own neighbours, and clusters of different densities come out together.

The price is dimension. As the number of variables grows, distances concentrate and neighbour lists start to overlap for no reason, so this method will report structure that is an artefact of the width of the table. `clusters$silhouette` is the check worth making, and agreement with `cluster_dbscan()` on the same points is worth more than either alone.

Noise

A point that never joined a dense part of the graph gets cluster `0`, has no silhouette and no row in `clusters`; `design$n_noise` is the count. Every point being noise is a possible answer and means the overlap asked for is not there, which is usually `eps` or `min_pts` being too large for the `k` in use.

See Also

`cluster_dbscan()`, the other density method here, which measures closeness with one radius rather than by shared neighbours.

Examples

```
## Two blobs of different spread, which is the case a single radius handles
## badly and shared neighbours handle well.
set.seed(1)
blobs <- rbind(
```

```
  matrix(rnorm(80, mean = -6, sd = 0.4), ncol = 2),
  matrix(rnorm(80, mean =  6, sd = 2.5), ncol = 2)
)
colnames(blobs) <- c("x", "y")
res <- cluster_snn(blobs, scale = FALSE, k = 10)
res
table(res$assignments$cluster)

plot(blobs, col = res$assignments$cluster + 1L, pch = 16,
     main = "cluster 0, in black, is noise")

## On a planted two-group structure, with everything left to be derived.
sim <- simulate_two_groups(n_feats = 30, n_up = 5, n_down = 5, seed = 3)
by_samp <- cluster_snn(sim$args$data)
table(cluster = by_samp$assignments$cluster, group = sim$args$group)
```

`coef.sa_fit` *Coefficients, summary and predictions from the model inside a fit*

Description

The `fit` element of an `sa_model` is the engine object, kept so that the model can `predict()`. These methods make the rest of what anyone does with a fitted model work on it as well: `coef()` and `summary()` read through to the `stats::lm()` or `stats::glm()` fit that `caret::train()` built, so what comes back is what the same call on that model would give.

Usage

```
## S3 method for class 'sa_fit'
coef(object, ...)

## S3 method for class 'sa_fit'
summary(object, ...)

## S3 method for class 'sa_fit'
predict(object, newdata = NULL, type = "raw", ...)
```

Arguments

`object`	The `fit` element of a `fit_linear_regression()`, `fit_logistic_regression()`, `fit_elastic_net()`, `fit_rf()` or `fit_svm()` result.
`...`	Passed on to the method of the underlying fit.
`newdata`	Rows to predict as `caret` takes them, which for a penalized fit or a machine is the design matrix rather than the predictor frame, or `NULL` for the rows the model was fitted on. `predict.sa_model()` takes the frame.
`type`	`"raw"` for the prediction as `caret` gives it, a fitted value or a class label; `"response"` for the same thing on the scale of the outcome, which for a classification is the probability of `outcome_lv[2]`; or `"prob"` for one column per class.

Details

The numbers are also in `$coefficients`, as a data.frame with the confidence interval beside them and, for a logistic regression, the odds ratio, and that table is what `coef.sa_model()` gives for the result as a whole. These methods are for when an `lm`-shaped answer is what is wanted instead: a named vector to index, or the summary layout that is already familiar.

On a `fit_elastic_net()` fit, `coef()` is the coefficients at the penalty that was chosen, which is the model the result describes and the one `predict()` predicts from, rather than the whole `lambda` path `glmnet` keeps. `summary()` has nothing of the same shape to return there, since a penalized fit has no standard errors and no residual degrees of freedom to test against; what it does have is in `$coefficients` and `$fit_stats`.

A `fit_rf()` or `fit_svm()` fit has neither. `coef()` on one is an error naming the importance table to read instead, rather than the `NULL` `coef()` on a `randomForest` object gives or the S4 indexing error it gives on a `ksvm` one, and `summary()` reaches a model with no summary method of its own. `print(fit$fit)` is the readable answer for both, and everything the result reports is in `$coefficients` and `$fit_stats`.

`predict()` here takes `newdata` as `caret` prepared it rather than as the fit received it, which for `fit_elastic_net()` and `fit_svm()` is a design matrix and not a data frame of predictors. `predict.sa_model()` is the method that takes the rows themselves, since the result is what knows which columns they were and what levels its factors had.

`predict()` is otherwise `caret`'s, with one word added. `caret::train()` knows `type = "raw"` for the prediction itself and `type = "prob"` for the class probabilities, and refuses anything else; `"response"`, which is what `stats::glm()` and `stats::lm()` call the prediction on the scale of the outcome, is therefore an error on an object that is plainly a logistic regression. It is accepted here and means the same thing in both models: the predicted value for a linear regression, and for a logistic one the probability of `outcome_lv[2]`, the class every coefficient and odds ratio in the result already describes. The probability is `caret`'s own, the second column of `type = "prob"`, so nothing is computed twice.

Value

From `coef()`, a named numeric vector of coefficients, `NA` for a term the fit could not estimate and exactly 0 for one a penalty dropped. From `summary()`, the `summary.lm` or `summary.glm` object of the underlying fit. The `Call` it prints belongs to `caret` rather than to you: `.outcome ~ .` on the predictor frame for a linear model, and empty for a logistic one, which `caret` fits without a formula. What was fitted to what is what `print()` on the result says.

From `predict()`, whatever `caret::predict.train()` returns for that `type`, except that `type = "response"` on a classification gives an unnamed numeric vector of probabilities, one per row of `newdata`.

Examples

```
fit <- fit_linear_regression(mtcars, outcome = "mpg",
                             predictors = c("wt", "hp"), cv = FALSE)
coef(fit$fit)
summary(fit$fit)

## The same numbers the result reports in its own table.
```

```
fit$coefficients

## On a classification, `type = "response"` is the probability of the second
## level of `outcome_lv`, the one the coefficients describe.
iris2 <- iris[iris$Species != "setosa", ]
clf <- fit_logistic_regression(iris2, outcome = "Species",
                               predictors = "Petal.Length",
                               outcome_lv = c("versicolor", "virginica"),
                               cv = FALSE)
head(predict(clf$fit, newdata = iris2, type = "response"))

## A penalized fit answers with the coefficients at the chosen penalty, the
## ones `$coefficients` reports, rather than with the whole lambda path.
pen <- fit_elastic_net(mtcars, outcome = "mpg", penalty = "lasso",
                       lambda = 0.5, cv = FALSE)
coef(pen$fit)
```

`coef.sa_model` *Coefficients of a fitted model*

Description

The coefficient table, `object$coefficients` itself: one row per term, in the order of `object$terms`, carrying everything the model estimated about each of them rather than the estimate alone.

Usage

```
## S3 method for class 'sa_model'
coef(object, ...)
```

Arguments

`object`	A fitted model, as returned by `fit_linear_regression()`, `fit_logistic_regression()`, `fit_elastic_net()`, `fit_rf()` or `fit_svm()`.
`...`	Ignored, present for consistency with `coef()`. `stats::coef()`'s own `complete` argument is among what is ignored: it drops the `NA` cells of a vector, which on a table would mean indexing a data.frame with a logical matrix, and the rows are what say which terms the model has.

Details

The `lm`-shaped answer, a named numeric vector, is `coef(x$fit)`. Two objects are reached by two calls, so they answer in their own terms rather than both giving the same thing: the engine object answers as the engine does, and the result object answers with the table it was assembled to hold. Ask `$fit` when a vector is what is wanted to index or multiply.

What the table holds depends on the model, and the columns say which kind it is. An unpenalized fit carries the standard error, the statistic, the p-value and the confidence limits, and a classification

the odds ratio; a penalized one has no standard error to report and carries `selected` instead, so `is.null(coef(x)$pval)` is the test for a fit that cannot be asked for inference. A forest and a support vector machine have no coefficient of any kind, and their table is the importance of each term rather than an effect per unit of it; see `fit_rf()` and `fit_svm()`. Every term keeps its row either way — a term a penalty dropped with an `estimate` of exactly 0, one the engine could not estimate with `NA` — since a table shorter than the model would be a table that had lost terms rather than one describing terms that went to zero.

Value

The `coefficients` element of `object`, a data.frame of one row per term.

See Also

`fit_linear_regression()`, `fit_elastic_net()`, `fit_rf()`, `fit_svm()`, and `coef.sa_fit()` for the same question asked of the engine object in `$fit`.

Examples

```
fit <- fit_linear_regression(mtcars, outcome = "mpg",
                             predictors = c("wt", "hp"), cv = FALSE)
coef(fit)

## The named vector is one object further in.
coef(fit$fit)

## A penalized fit answers with what it has: no inference, but `selected`.
pen <- fit_elastic_net(mtcars, outcome = "mpg", penalty = "lasso",
                       lambda = 0.5, cv = FALSE)
coef(pen)
```

---

compare_categorical_groups

Test a contingency table with every applicable test at once

Description

Crosses two categorical variables into a contingency table and returns an asymptotic and an exact test of it side by side, together with the measures of how strong the association is. Nothing is chosen on the user's behalf: reporting both makes disagreement between them visible, which is the situation where the choice between an approximation and an exact enumeration actually matters.

Usage

```
compare_categorical_groups(
  data,
  category_lv = NULL,
```

```
  control_label = NULL,
  paired = FALSE,
  conf_level = 0.95,
  correct = TRUE,
  exact = NULL,
  simulate_p_value = FALSE,
  n_resamples = 9999,
  max_levels = 20L,
  seed = NULL,
  diagnose = TRUE
)
```

Arguments

data	A data.frame (or matrix) whose columns are the categorical variables. Unlike the other comparison scenarios there is no `feats` argument: the columns are what is tested rather than what is measured, so a factor, a character vector, a logical or a 0/1 coded column all read as categorical and are taken as their labels.
category_lv	Named list giving the levels of each variable, with the reference level first, or `NULL` to take every column of `data` with its levels in sorted order. Naming it does three things sorting cannot: it picks which columns take part, it fixes which level is the reference the odds ratio is read against, and it drops the rows belonging to any level it leaves out. Every level it names has to be present in `data`.
control_label	The level to hold as the reference. For an independent design this is one name per variable it points at, as a named list (`list(smoker = "n", grade = "low")`) or as a named character vector (`c(smoker = "n")`); the two shapes say the same thing, and a variable it says nothing about is left as it arrived. A matched design has one level set shared by every condition, so there it is a single level name. Defaults to `NULL`, the first level of every variable, which under `category_lv = NULL` is the first in sorted order.
paired	Logical. If `TRUE`, the columns are read as repeated conditions measured on the same row rather than as different variables. See "Which tests run".
conf_level	Confidence level for the association intervals and for the conditional odds ratio of Fisher's exact test.
correct	Whether to apply the continuity correction to the chi-square approximation. `stats::chisq.test()` applies it to 2 x 2 tables only, so it changes nothing on a larger one, and the exact branch of McNemar's test needs none.
exact	Read only by McNemar's test. `TRUE` for the exact binomial test on the discordant pairs, `FALSE` for the chi-square approximation, or `NULL` to take the exact test when there are fewer than 25 discordant pairs. `parameters$exact` records which one ran.
simulate_p_value	Read only by the tests of an independent design. Replaces the chi-square approximation with a Monte Carlo p-value, and is the way to get an answer out of Fisher's test on a large r x c table that cannot be enumerated.
n_resamples	How many tables the Monte Carlo p-value is taken over.

`max_levels` How many levels a variable may take before it is refused as a category. `as.character()` turns a continuous measurement into one label per observation, which makes a table with a cell per row and no test of association to run on it; this is the ceiling that catches that rather than letting it through. Checked against the levels actually used, so naming three levels of a fifty-valued column in `category_lv` is a way through. Raise it for a variable that genuinely has this many.

`seed` Seed for the Monte Carlo p-value, restored on exit, so a simulated p-value is reproducible without the caller's random stream being disturbed.

`diagnose` Logical. If `TRUE`, the rule the reported approximation rests on is attached as `$diagnostics`.

Details

This is the one scenario in the package with no feature axis. Every other comparison asks its question once per numeric column and returns a table with one row per column; a contingency table is asked about as a whole, so the result carries the table itself in `$cells` and one row per test in `$tests`. That is also why the result is not an `sa_comparison` and does not go through `estimate_significance()`: a volcano plot needs a signed effect per feature and this scenario has no feature axis to carry one. `estimate_categorical_significance()` is the counterpart that reads it, one verdict per cell. See "Why this is not a comparison result".

A row is dropped for one of two reasons, and the two are counted apart. A row naming a level `category_lv` leaves out was measured and excluded, and appears as `design$n_dropped`. A row missing a value in any variable was not measured, and appears as `design$n_incomplete`; a table needs the whole row, so the deletion is listwise.

The warning `stats::chisq.test()` raises about small expected counts is not passed on. `$diagnostics` states the same fact as a number, and the exact test that does not need the approximation is already in the same result, so a warning would be a less precise version of what is sitting next to it.

Fisher's exact test enumerates every table with the observed margins, and on a large r x c one there are more of those than the algorithm's workspace holds. That is a limit of the enumeration rather than a fault in the data, so `$tests$fisher_test$pval` comes back `NA` with a message saying so, instead of the whole call failing and taking the chi-square result with it. `simulate_p_value = TRUE` is the way to get an answer there.

Value

An `sa_categorical` object: a plain list, so it survives being written out as JSON and read back in another language, with an S3 class on top that supplies `print()`, `plot()` and `as.table()`. Its elements are

`analysis` `"categorical_comparison"`.

`variables` The variable names, in the order `category_lv` fixed.

`design` `category_lv`, `null`, `paired`, `pairing`, `dim`, `row_var`, `col_var`, `n_used`, `n_dropped` and `n_incomplete`. `null` is `"independence"`, `"symmetry"` or `"marginal_homogeneity"`, and it is what `expected` and the residuals are read under.

`parameters` The analysis choices as used, so `exact` says which branch of McNemar's test ran rather than what was passed.

`cells` One row per cell of the table: `row_level`, `col_level`, `observed`, `expected`, `residual`, `std_residual`, `prop_total`, `prop_row` and `prop_col`. This is the canonical form of the table, and what `draw_mosaic_plot()` reads. `as.table()` folds it back into a `table()`, which is the shape to read it in rather than a second copy to keep.

`tests` One one-row data.frame per test, named `chisq_test`, `fisher_test`, `mcnemar_test` or `cochran_q`, each carrying `n_used`, `statistic`, `df`, `pval`, `lower_conf` and `upper_conf`. There is no `pval_adj`.

`test_info` The method `id`, a readable `label` and whether the test is a matched one, per element of `tests`.

`association` One row per measure: `measure`, `estimate`, `lower_conf`, `upper_conf`. Which measures are defined depends on the design and on the size of the table.

`diagnostics` The approximation rule this design rests on, or `NULL`.

`metadata` `package_version`, `r_version`, `platform` and an ISO-8601 `timestamp`.

One test, three questions

The chi-square statistic answers what look like three different questions, and the arithmetic is the same in each.

Independence Both variables were measured on one sample: is there any association between them? This is the reading the default label uses.

Homogeneity One variable says which sample a row came from and the other is what was measured: do the samples share the same distribution over the categories? The sampling scheme differs, the null hypothesis is stated differently, and the test is the same one.

Goodness of fit One variable against a set of expected proportions. This one is genuinely different, being about a single margin rather than a cross-classification, and it is not implemented here.

Which of the first two a caller means is a fact about their sampling scheme, not about their data, and the data cannot be inspected to find out. So this function does not ask, and both readings are the same call.

Which tests run

Decided by `paired` and by how many variables `category_lv` names, so there is no argument naming a test.

design	null hypothesis	tests
independent, two variables	independence	Chi-square test of independence, Fisher's exact test
matched, two binary conditions	symmetry	McNemar's test
matched, three or more binary conditions	marginal homogeneity	Cochran's Q test

A matched design reads the columns as repeated measurements of one thing on the same row, so pairing is by row and there is no `id` argument. It also requires the levels to be binary: the tests of symmetry that generalise McNemar's test past two levels, Bowker's and Stuart-Maxwell's, are not implemented, and a three-level matched design is an error naming them rather than a table quietly collapsed to two levels.

What the cells are expected to hold

`design$null` names the hypothesis the whole result is about, and `$cells$expected` is read under it. This matters because a contingency table can be held against more than one null and the three designs above hold it against three different ones.

Under **independence** a cell is expected at the product of its margins over the total. Under **symmetry** it is expected at the average of it and its transpose, so the diagonal is expected at exactly what it holds and carries no residual: a pair that answered the same way under both conditions says nothing about which condition raises the response. Under **marginal homogeneity** every condition is expected to show the pooled response rate, which on a condition-by-response table is the same arithmetic as independence and a different claim about the world.

The residuals follow the same hypothesis, which is what keeps them and the p-value beside them talking about one thing. In a matched 2 x 2 the squared Pearson residuals of the two discordant cells sum to `(b - c)^2 / (b + c)`, which is McNemar's uncorrected statistic exactly; the cell table and the test are the same arithmetic read two ways. `draw_mosaic_plot()` shades on those residuals, so the picture is about the hypothesis the result tested rather than about independence in every case.

`std_residual` is the exception. Its variance correction is derived for a table held against its own margins and has no counterpart under symmetry, so it is `NA` there rather than a number that looks referable to a standard normal and is not.

Why this is not a comparison result

`sa_new_comparison()` holds every table in a comparison to one row per feature, and `estimate_significance()` reads one `log2fc` per row off that axis. Neither exists here. There is one table, so there is one p-value and no multiplicity to adjust across, which is why no `pval_adj` column is carried. And an association has no sign: `cramers_v` says how far the table sits from independence but not in which direction, because past a 2 x 2 table there is no single direction to name. On a 2 x 2 table there is one, and `odds_ratio` reports it.

The result is therefore an `sa_categorical`, which keeps the vocabulary of a comparison – `design`, `parameters`, `tests` beside `test_info`, `metadata` – without claiming a contract it cannot meet. `draw_mosaic_plot()` is what reads it, in the place `draw_volcano_plot()` holds for the numeric scenarios.

Every sentence above is about the table. A **cell** does have both axes: `observed / expected` is a signed departure defined on a table of any shape, and `std_residual` beside it is referred to a standard normal, so the cells of one table are a family to adjust across. `estimate_categorical_significance()` is what reads that axis, and it is a function of `$cells` alone, so no slot is carried for it.

Direction

Set once, by the order of the levels in `category_lv`, and `control_label` is the second way of stating it. Every quantity that has a direction follows it.

`odds_ratio` is above 1 when the **second** level of each variable occurs with the second level of the other more often than independence would give, and `phi_coefficient` is above zero in the same situation. Pointing `control_label` at the other level of either variable inverts both, and pointing it at the other level of both leaves them where they were. In a matched design the second level is the response the conditions are compared on, so `odds_ratio_paired` is above 1 when the later condition raises it.

Which measures are defined

An independent table always reports `cramers_v` and `contingency_coefficient`, and a 2 x 2 one adds `phi_coefficient` and `odds_ratio`. A matched 2 x 2 reports `odds_ratio_paired`, `risk_difference_paired` and `cohens_g`, all three read off the discordant cells. Three or more matched conditions report `kendalls_w`.

Only `odds_ratio` and the paired measures carry an interval. The others are functions of the chi-square statistic whose sampling distribution has no closed-form interval, so their `lower_conf` and `upper_conf` are `NA`: the columns exist because every result table in the package carries them, not because every number in them is finite.

Every measure is built from the **uncorrected** chi-square statistic, whatever `correct` was set to. Yates' correction is about referring a discrete statistic to a continuous distribution, which is a statement about a p-value; letting it into an effect size would make the reported strength of an association depend on a choice made about its tail probability.

References

Pearson, K. (1900). On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine*, 50(302), 157-175.

Fisher, R. A. (1935). The logic of inductive inference. *Journal of the Royal Statistical Society*, 98(1), 39-82.

McNemar, Q. (1947). Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2), 153-157.

Cochran, W. G. (1950). The comparison of percentages in matched samples. *Biometrika*, 37(3-4), 256-266.

Agresti, A. (2002). *Categorical Data Analysis*, 2nd ed. Wiley.

See Also

`simulate_categorical_groups()` for a table whose association is known, `estimate_categorical_significance()` to reduce the result to a verdict per cell, `draw_mosaic_plot()` to draw it, and `compare_multiple_groups()` for the case where the measurement is numeric and only the grouping is categorical.

Examples

```
## Both columns are categorical, so `data` is the whole input.
smoking <- data.frame(
  smoker = rep(c("y", "n"), each = 60),
  grade  = c(rep(c("high", "mid", "low"), c(10, 20, 30)),
             rep(c("high", "mid", "low"), c(30, 20, 10)))
)
res <- compare_categorical_groups(smoking)
res

## The table itself, and the cells that made the statistic what it is.
as.table(res)
res$cells[c("row_level", "col_level", "observed", "expected", "std_residual")]
```

```
## How strong the association is, which no test reports.
res$association

## `category_lv` picks the levels and their order, and drops the rest. Two
## levels of `grade` make a 2 x 2 table, which is where the odds ratio exists.
two_by_two <- compare_categorical_groups(
  smoking,
  category_lv = list(smoker = c("n", "y"), grade = c("low", "high"))
)
subset(two_by_two$association, measure == "odds_ratio")

## `control_label` restates the reference without the levels being retyped,
## and pointing it at one variable inverts the odds ratio.
flipped <- compare_categorical_groups(
  smoking,
  category_lv   = list(smoker = c("n", "y"), grade = c("low", "high")),
  control_label = c(smoker = "y")
)
subset(flipped$association, measure == "odds_ratio")

## A matched design: the columns are the same question asked twice, so pairing
## is by row, the null is symmetry rather than independence, and McNemar's test
## reads only the discordant pairs.
before_after <- data.frame(
  before = rep(c("pass", "fail"), c(20, 30)),
  after  = c(rep(c("pass", "fail"), c(18, 2)), rep(c("pass", "fail"), c(14, 16)))
)
matched <- compare_categorical_groups(before_after, paired = TRUE)
matched$design$null

## Which is what the cell table is read under: the diagonal is expected at
## exactly what it holds, so only the discordant cells carry a residual.
matched$cells[c("row_level", "col_level", "observed", "expected", "residual")]

## A simulated table hands over exactly the arguments this function takes.
sim <- simulate_categorical_groups(n_samples = 400, assoc = 0.4, seed = 1)
fit <- do.call(compare_categorical_groups, sim$args)
cbind(planted = sim$truth$cramers_v,
      estimated = fit$association$estimate[1])
```

`compare_factorial_groups` *Analyse a crossed-factor design as one model*

Description

The factorial counterpart of `compare_multiple_groups()`, and the one place in the package where a scenario function fits **one** model rather than running every applicable test side by side. Two crossed factors are a two-way ANOVA, three are a three-way ANOVA and more than three are a

factorial ANOVA, and those are three names for the same fully crossed linear model rather than three procedures to choose between. Which name applies follows from `length(factor_lv)`, so there is no argument for it; the answer is reported in `design$anova_type` and in the readable label of the test.

Usage

```
compare_factorial_groups(
  data,
  feats,
  factors,
  factor_lv = NULL,
  control_label = NULL,
  within = NULL,
  id = NULL,
  conf_level = 0.95,
  ss_type = c("III", "II", "I"),
  posthoc = TRUE,
  posthoc_alpha = 0.05,
  posthoc_scope = c("both", "marginal", "simple"),
  fc_mean = c("arith", "geom"),
  input_scale = c("raw", "log2"),
  p_adjust = "BH",
  diagnose = TRUE
)
```

Arguments

`data`	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
`feats`	Character vector of numeric column names in `data` to test. One row of `$tests` and `$effect` per entry, and one row of `$terms` per entry and term.
`factors`	Named list of the crossed factors, each entry either the name of a column of `data` or a vector with one entry per row of it. Its length is how many factors are crossed, and there have to be at least two: one factor is `compare_multiple_groups()`. The **first factor is the primary one** and the order of the list is the order the terms are listed in, which is also the order Type I sums of squares are taken in.
`factor_lv`	Named list giving the levels of each factor, with the reference level first, or `NULL` to take the levels from the data in sorted order. Naming it does two things sorting cannot: it fixes which level is the reference that `effect` and the cell labels are read against, and it drops the rows belonging to any level it leaves out. Every level it names has to be present in `factors`. `control_label` fixes the reference on its own, for when that is all there is to say.
`control_label`	The level to hold as the reference, one name per factor it points at, as a named list (`list(treatment = "control", sex = "male")`) or as a named character vector (`c(treatment = "control")`). The two shapes say the same thing. The level it names moves to the front of its own factor and the rest keep the order they were given, so the reference cell is the one where every factor sits at the level

named here and every fold change is read against it. A factor it says nothing about is left as it arrived, which is what makes pointing one factor of three a sentence rather than a rewrite of all three. Defaults to NULL, the first level of every factor, which under factor_lv = NULL is the first in sorted order.

within — Names of the factors measured within subjects. **Not implemented in this version**: a non-empty value is an error rather than a design that is silently analysed as though the repeated measurements were independent. The argument exists so that the error names what is missing, which an unused argument (within) from do.call() would not.

id — Subject identifier with one entry per row of data. Used only by a within-subject design, so it is ignored with a warning here.

conf_level — Confidence level for the post-hoc intervals.

ss_type — Which sums of squares the term tests are built from, "III", "II" or "I". They agree on a balanced design and differ on an unbalanced one; see "Unbalanced cells" below.

posthoc — Logical. If FALSE, no pairwise stage runs and the result carries no $posthoc slot.

posthoc_alpha — A contrast is computed when the term it belongs to has a pval_adj at or below this value. Set it to 1 to compare every level of every factor regardless of the term tests.

posthoc_scope — Which contrasts the pairwise stage covers: "marginal" for the comparisons a main effect is about, "simple" for the same comparisons inside one combination of the other factors, or "both".

fc_mean — Which centre the fold change divides, "arith" for the arithmetic mean or "geom" for the geometric mean. Defaults to "geom" when input_scale = "log2" and to "arith" otherwise.

input_scale — The scale data arrives on, "raw" or "log2". On the log2 scale each observation is raised back through 2^x before the centres are taken, so the ratios mean what they do for raw input. This changes the effect table only, never the tests. See compare_two_groups() for the full account.

p_adjust — Multiplicity adjustment passed to stats::p.adjust(), applied along the feature axis. Use "none" to disable. In $terms it is applied within each term rather than over the whole table, for the reason given under "Two axes" below.

diagnose — Logical. If TRUE, the normality, homogeneity of variance and outlier checks the model rests on are attached as $diagnostics, taken over the cells.

Details

What multiplies instead is the number of answers. One factor asks whether its levels are alike; crossing a second one asks that of each factor and asks whether either one's effect depends on the other, and those questions fail separately. A single p-value cannot carry them, so the result has an axis the sibling functions do not: $terms, one row per feature and model term.

Features that cannot be tested do not abort the run. Their rows are filled with NA and all such features are reported together in a single warning.

Value

A sa_comparison object with the layout compare_multiple_groups() returns, minus $pairwise and plus $terms and $cells. Its elements are

analysis "factorial_comparison".

features The feature names, in the row order every table uses.

design factor_lv, anova_type, n_factors, group_lv (the cell labels), cell_n, n_empty_cells, paired, pairing, within, n_dropped and unmatched_ids.

parameters The analysis choices as used, plus n_posthoc, the number of features that entered the pairwise stage.

effect One row per feature: n_used, n_cells, ref_center, extreme_cell, extreme_center, fold_change and log2fc. The reference cell is the one where every factor sits at its first level, which factor_lv or control_label states, and extreme_cell is whichever cell sits furthest from it on the log2 scale, the same definition simulate_factorial_groups() records in truth$extreme_cell.

tests One table, anova_test, holding the whole-model F test with one row per feature.

terms One row per feature and model term, holding features, terms, term_order, n_used, df, ss, ms, f_stat, df_error, eta_sq, partial_eta_sq, log2_effect, pval and pval_adj.

cells One row per feature and cell of the grid, holding features, one column per factor named after it and holding the level name, cell (the dot-joined label, as in design$group_lv), n, mean, sd and se. mean is the arithmetic cell mean the model was fitted on and se is sqrt(ms_error / n), pooled over the whole model, so the variance of a marginal mean over a set of cells is recoverable from it. This is the table draw_interaction_plot() reads, and the one part of the grid the rest of the result cannot be read backwards into.

posthoc anova_test, one row per feature and contrast, carrying the post-hoc contract columns plus factor and stratum. Absent when posthoc = FALSE.

test_info The method id, a readable label naming the ANOVA that ran, and the post-hoc procedure that followed it.

diagnostics Assumption checks over the cells, or NULL.

metadata package_version, r_version, platform and an ISO-8601 timestamp.

Two axes

The whole-model test in $tests$anova_test asks whether a feature responds to the design at all. That is one question per feature, and it is exactly the one-way ANOVA that treats the cells as groups: a fully crossed model is the cell means model written in another basis, so the two fit the same values and leave the same residuals. Keeping it there is what lets estimate_significance(), print(), draw_forest_plot() and draw_volcano_plot() read a factorial result without knowing that it is one.

Which **part** of the design a feature responds to is the question a crossed model was fitted to answer, and it has one answer per term. Those live in $terms, whose terms and term_order columns are the ones simulate_factorial_groups() writes into truth_term, so a result and an answer key merge on c("features", "terms") with neither side renamed.

pval_adj is adjusted within each term, across features. A term is one family: asking of five hundred features whether the treatment matters is five hundred instances of one question, while asking

whether the treatment matters and whether it depends on sex are two questions and pooling them would correct each for the other's multiplicity.

There is no `$pairwise` slot. Its keys are contrast labels, and a factorial design has a two-dimensional contrast axis, factor by stratum, that a flat list keyed by label cannot hold without a naming convention. Until there is one, `estimate_significance(by = "contrast")` reports that the result has no pairwise stage, which is the true statement.

The size of a term

`eta_sq` and `partial_eta_sq` say how much of the variance a term accounts for, and neither has a sign, so neither can be read as a direction. `log2_effect` can: it is the largest ANOVA component of the term, with its sign, taken by decomposing `log2()` of the same cell centres `effect` is built from. It is what `estimate_significance()` puts on the effect axis in a term reading, and so what `draw_volcano_plot()` plots a term panel against when passed the verdict of `estimate_significance(fact, by = "term")`.

A component is a deviation from what the rest of the model already predicts, **not** a difference between two levels. A two-level factor whose levels differ by one log2 unit has components `-0.5` and `+0.5`, so `log2_effect` is `0.5` where the marginal fold change is 2. The definition is kept because it is the one `simulate_factorial_groups()` records in `truth_term$max_abs_delta`, which makes the column scorable against an answer key; the consequence is that cutoffs meant for a fold change, `log2fc_cutoff = 1` among them, are stricter here than they look.

Components do not depend on which cell is the reference, since subtracting one cell from every cell cancels out of the deviation. That is the one place the term axis is simpler than `effect$log2fc`, which is read against the reference cell by definition.

Unbalanced cells

With equal cell sizes the three types of sums of squares are identical and `ss_type` changes nothing. They part company when the cells are unequal, because the factors are then no longer orthogonal and some of the variation can be attributed to more than one of them.

The default is **Type III**, each term adjusted for every other term. It is the type whose main effects are statements about the levels rather than about how many observations happened to land in each, which is what makes it agree with the unweighted marginal means the post-hoc stage compares and with the decomposition `simulate_factorial_groups()` plants. **Type II** adjusts a term for every term that does not contain it, leaving a main effect unadjusted for the interaction it is part of. **Type I** is sequential and therefore depends on the order `factors` was written in; it is what `stats::aov()` reports, which makes it the type to ask for when checking these numbers against an external implementation on unbalanced data.

A cell holding no usable observation leaves the crossed model with nothing to estimate there, so that feature's rows are `NA` and the reason is reported in one warning. `design$n_empty_cells` counts the cells that hold no rows at all.

Marginal contrasts and simple effects

A marginal contrast compares two levels of one factor with the other factors averaged away, which is the comparison a main effect is a statement about. A simple effect compares the same two levels inside one combination of the other factors, and it is the only one of the two that means anything when an interaction is real: if the treatment helps males and harms females, the average of the two is

a number that describes nobody. `stratum` tells them apart, `NA` for a marginal contrast and the levels of the other factors joined by a dot for a simple one, and `factor` names the factor being compared. Both columns are the ones `truth_contrast` carries.

The marginal mean is the **unweighted** mean of the cell means rather than the mean of the observations, so a level is not pulled towards whichever combination of the other factors was sampled most heavily. Every contrast is scaled by the mean square error of the whole model, which is what makes the pairwise stage consistent with the term tests instead of a second analysis of the same data, and judged against the studentised range over the number of levels of its own factor. Those p-values are already family-wise within a block, so there is no `posthoc_p_adjust` argument to apply and `parameters$posthoc_p_adjust` is `NA`.

Which contrasts are computed is decided term by term. A marginal contrast of a factor runs when that factor's main effect cleared `posthoc_alpha`, and a simple effect runs when the interaction of the factor with the factors held fixed cleared it. Gating everything on the whole-model test instead would compare the levels of a factor that the model says has no effect, on the strength of a different factor that has one.

References

Fisher, R. A. (1935). *The Design of Experiments*.

Yates, F. (1934). The analysis of multiple classifications with unequal numbers in the different classes. *Journal of the American Statistical Association*, 29(185), 51-66.

Tukey, J. W. (1949). Comparing individual means in the analysis of variance. *Biometrics*, 5(2), 99-114.

Kramer, C. Y. (1956). Extension of multiple range tests to group means with unequal numbers of replications. *Biometrics*, 12(3), 307-310.

See Also

`compare_multiple_groups()` for a single factor, `simulate_factorial_groups()` for data whose answer per term is known, and `draw_forest_plot()` to draw the result.

Examples

```
## Two crossed factors, so a two-way ANOVA. The factors name columns of `data`.
res <- compare_factorial_groups(
  data    = warpbreaks,
  feats   = "breaks",
  factors = list(wool = "wool", tension = "tension")
)
res

## The whole-model test, one row per feature: does this feature respond at all.
res$tests$anova_test

## The answer per term, which is what the design was crossed to get.
res$terms

## Every fold change is read against the cell where each factor sits at its
## first level. `control_label` moves one of those levels without the rest
```

```
## having to be listed, so the reference cell becomes A.M rather than A.L.
medium <- compare_factorial_groups(
  data          = warpbreaks,
  feats         = "breaks",
  factors       = list(wool = "wool", tension = "tension"),
  control_label = list(tension = "M"),
  posthoc       = FALSE
)
medium$design$group_lv[1]
medium$effect[c("ref_center", "extreme_cell", "log2fc")]

## The whole-model volcano names the reference cell on the x axis.
draw_volcano_plot(estimate_significance(res, log2fc_cutoff = 0.1))

## Marginal contrasts and simple effects, told apart by `stratum`.
subset(res$posthoc$anova_test, factor == "tension" & is.na(stratum))

## A simulated design hands over exactly the arguments this function takes,
## and `truth_term` is the answer key for `$terms`.
sim <- simulate_factorial_groups(n_feats = 12, n_per_cell = 8, seed = 1)
fac <- do.call(compare_factorial_groups, sim$args)
scored <- merge(fac$terms, sim$truth_term, by = c("features", "terms"))
with(scored, table(called = pval_adj <= 0.05, planted = is_effect, terms))
```

`compare_multiple_groups` *Run every applicable multi-group test at once*

Description

Compares three or more group levels across any number of numeric features and returns the omnibus tests side by side, each followed by the post-hoc procedure that shares its assumptions. As with `compare_two_groups()`, nothing is chosen on the user's behalf: reporting the parametric, the rank-based and the robust result together makes disagreement between them visible, which is the situation where the choice of test actually matters.

Usage

```
compare_multiple_groups(
  data,
  feats,
  group,
  group_lv,
  control_label = group_lv[1],
  id = NULL,
  paired = FALSE,
  conf_level = 0.95,
  tr = 0.2,
```

```
  posthoc = TRUE,
  posthoc_alpha = 0.05,
  fc_mean = c("arith", "geom"),
  input_scale = c("raw", "log2"),
  p_adjust = "BH",
  posthoc_p_adjust = "holm",
  diagnose = TRUE
)
```

Arguments

data	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
feats	Character vector of numeric column names in data to test. One output row per entry.
group	Grouping vector with one entry per row of data.
group_lv	Character vector of at least three group levels. The first is the reference the effect table is expressed against and the level every post-hoc contrast subtracts, so a contrast reads treat_1 - control rather than the other way round. Rows belonging to any other level are dropped.
control_label	The level to hold as the reference. Naming it moves that level to the front of group_lv and leaves the rest in the order given, so the fold change and every post-hoc contrast are expressed against it without the levels having to be retyped. Defaults to group_lv[1], the reference the order already asked for.
id	Subject identifier with one entry per row of data. Required when paired = TRUE and ignored otherwise.
paired	Logical. If TRUE, the levels of group are treated as repeated conditions measured on the subjects named by id.
conf_level	Confidence level for the post-hoc intervals.
tr	Trimming proportion for the trimmed mean ANOVA, in [0, 0.5). Ignored when paired = TRUE.
posthoc	Logical. If FALSE, no pairwise stage runs and the result carries no $posthoc or $pairwise slot.
posthoc_alpha	A feature enters the post-hoc stage when its omnibus pval_adj is at or below this value. Set it to 1 to compare every feature regardless of its omnibus result.
fc_mean	Which centre the fold change divides, "arith" for the arithmetic mean or "geom" for the geometric mean. Defaults to "geom" when input_scale = "log2" and to "arith" otherwise.
input_scale	The scale data arrives on, "raw" or "log2". On the log2 scale each observation is raised back through 2^x before the centres are taken, so the ratios mean what they do for raw input. This changes the effect table only, never the tests. See compare_two_groups() for the full account.
p_adjust	Multiplicity adjustment applied across feats within each omnibus table, passed to stats::p.adjust(). Use "none" to disable.

`posthoc_p_adjust`
: Multiplicity adjustment applied across the contrasts within each feature of a post-hoc table. Ignored for Tukey's HSD and Games-Howell, whose p-values are already family-wise.

`diagnose`
: Logical. If `TRUE`, the normality, homogeneity of variance and outlier checks the tests rest on are attached as `$diagnostics`.

Details

Which family runs depends on `paired`:

slot	`paired = FALSE`	`paired = TRUE`
`anova_test`	One-way ANOVA	Repeated measures ANOVA
`welch_test`	Welch's ANOVA	not applicable
`robust_test`	Yuen's trimmed mean ANOVA	not applicable
`kruskal_test`	Kruskal-Wallis	Friedman

Each one is followed by its own post-hoc procedure, never by a borrowed one: one-way ANOVA by Tukey's HSD, Welch's ANOVA by Games-Howell, the trimmed mean ANOVA by pairwise Yuen tests, Kruskal-Wallis by Dunn's test, repeated measures ANOVA by pairwise paired t-tests and Friedman by Conover's test. A rank-based omnibus test is therefore never followed by a parametric comparison.

Direction is set once, by the order of `group_lv`, and `control_label` is the second way of stating that order: the level it names moves to the front and the rest keep the order they were given. The move carries further here than it does in `compare_two_groups()`, since the reference is the denominator of the fold change and the subtracted side of every post-hoc contrast at the same time, and the remaining levels keep contrasting each other in display order. Naming a level that `group_lv` already holds is a correction rather than a contradiction, which is why it is accepted here and refused by the models, whose `outcome_lv` holds the two classes and nothing else.

Features that cannot be tested do not abort the run. Their row is filled with `NA` and all such features are reported together in a single warning.

Value

A `sa_comparison` object with the same layout `compare_two_groups()` returns, plus the `posthoc` and `pairwise` slots that only a comparison of three or more levels has. Its elements are

`analysis` `"multi_group_comparison"`.

`features` The feature names, in the row order every table uses.

`design` `group_lv`, `paired`, `pairing`, `n_dropped` and `unmatched_ids`.

`parameters` The analysis choices as used, plus `n_posthoc`, the number of features that entered the pairwise stage per test.

`effect` One row per feature, described below.

`tests` The omnibus tables, one row per feature.

`posthoc` One table per omnibus test, one row per feature and pair of levels, carrying `features`, `contrast`, `group1`, `group2`, `n1`, `n2`, `estimate`, `stderr`, `statistic`, `df`, `pval`, `pval_adj`, `lower_conf` and `upper_conf`. Absent when `posthoc = FALSE`.

`pairwise` The same numbers one contrast at a time, described below. Absent when `posthoc = FALSE`.

`test_info` Per test, the method `id`, a readable `label` and the post-hoc procedure that followed it.

`diagnostics` Assumption checks, or `NULL`.

`metadata` `package_version`, `r_version`, `platform` and an ISO-8601 `timestamp`.

The `effect` table holds `n_used`, `n_groups`, `ref_center` (the centre of `group_lv[1]`), `extreme_level`, `extreme_center`, `fold_change` and `log2fc`. `extreme_level` is whichever level sits furthest from the reference on the log2 scale, and the ratio puts it over the reference, so a positive `log2fc` means that level is the higher one. `group_lv[1]` is the denominator here exactly as it is in `compare_two_groups()`: the first level is the reference in both. The post-hoc contrasts subtract the reference for the same reason, so a feature the treatment raised is positive in `effect$log2fc` and in its post-hoc `estimate` alike. Keeping the column named `log2fc` means `estimate_significance()` and `draw_volcano_plot()` work on a multi-group result unchanged.

Omnibus intervals

The omnibus tables carry `lower_conf` and `upper_conf` as required by the result contract, and both are `NA`. An omnibus test states that the levels are not all alike; it does not state by how much, and there is no single quantity for an interval to be about. The intervals of a multi-group comparison live in `$posthoc`, where each row is one contrast and does have a scale of its own.

Post-hoc stage

Only features whose omnibus `pval_adj` clears `posthoc_alpha` are compared pairwise, and a feature that did not qualify is absent from the post-hoc table rather than present with `NA`. An absent row means the question was never asked; an `NA` row means it was asked and could not be answered, and the two should not look the same. `parameters$n_posthoc` records how many features entered each stage.

Tukey's HSD and Games-Howell control the error rate over the whole set of contrasts through the studentised range, so `pval_adj` equals `pval` for those two and `posthoc_p_adjust` is not applied. Dunn, Conover, pairwise Yuen and pairwise paired t-tests are adjusted across the contrasts within each feature.

Reading one contrast at a time

`$posthoc` stacks every contrast of every feature into one long table, which is the honest record of what was asked but an awkward shape for a reader who came for a single comparison. `$pairwise` holds the same numbers keyed first by test and then by contrast, so `res$pairwise$anova_test[["virginica - setosa"]]` is that one comparison across all features.

Those tables are rectangular where `$posthoc` is ragged: each holds every feature, in the order `features` fixes, so contrasts can be lined up against each other and against the omnibus tables by position. A feature that did not qualify is present with its inference columns `NA`.

They add `fold_change` and `log2fc`, which no post-hoc procedure reports, being the ratio of the two group centres rather than anything a test produced. The ratio puts `group1` over `group2`, matching

the direction `estimate` reads in, so the two always agree in sign. Note that this is a different quantity from `effect$log2fc`, which compares the most extreme level rather than a named pair, though the two now point the same way: both divide by the reference. A feature that never entered the pairwise stage still has its ratio: dividing two centres does not require a test to have been run.

`estimate_significance()` reads these tables with `by = "contrast"`, whose `significance` element is then one verdict table per contrast rather than one table.

Repeated conditions

A within-subject omnibus test needs a complete rectangle, so `id` is required and subjects missing any condition are dropped whole rather than partially used. The number dropped is reported in `design$unmatched_ids`. Missing values are then handled per feature: a subject with `NA` on one feature is left out of that feature only, which is why `n_used` can differ between features.

Row order pairing, which `compare_two_groups()` allows, is deliberately not offered here. With two groups it is at least well defined; with three or more it would also have to assume every condition is stored in the same subject order, and there is no way to notice when it is not.

References

Welch, B. L. (1951). On the comparison of several mean values: an alternative approach. *Biometrika*, 38(3-4), 330-336.

Kruskal, W. H. and Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260), 583-621.

Tukey, J. W. (1949). Comparing individual means in the analysis of variance. *Biometrics*, 5(2), 99-114.

Games, P. A. and Howell, J. F. (1976). Pairwise multiple comparison procedures with unequal n's and/or variances: a Monte Carlo study. *Journal of Educational Statistics*, 1(2), 113-125.

Dunn, O. J. (1964). Multiple comparisons using rank sums. *Technometrics*, 6(3), 241-252.

Conover, W. J. (1999). *Practical Nonparametric Statistics*, 3rd edition.

See Also

`compare_two_groups()` for exactly two levels, `diagnose_distribution()` for the assumption checks on their own, and `draw_forest_plot()` to draw the result.

Examples

```
## Independent samples: all three iris species
feats <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")
res <- compare_multiple_groups(
  data     = iris,
  feats    = feats,
  group    = iris$Species,
  group_lv = c("setosa", "versicolor", "virginica")
)
res
res$tests$anova_test
res$tests$kruskal_test
```

```
## The pairwise stage, one row per feature and pair
head(res$posthoc$anova_test)

## The same numbers one contrast at a time, every feature present. Setosa is
## the reference, so it is the level every contrast subtracts.
names(res$pairwise$anova_test)
res$pairwise$anova_test[["virginica - setosa"]]

## Setosa is the reference, so a positive log2fc means the most extreme
## other species is the larger one.
res$effect

## Repeated conditions: each ChickWeight chick weighed at several times
chicks <- ChickWeight[ChickWeight$Time %in% c(0, 6, 12, 18), ]
rep_res <- compare_multiple_groups(
  data     = data.frame(weight = chicks$weight),
  feats    = "weight",
  group    = paste0("day", chicks$Time),
  group_lv = c("day0", "day6", "day12", "day18"),
  id       = chicks$Chick,
  paired   = TRUE
)
rep_res$tests$anova_test   # Mauchly, Greenhouse-Geisser and Huynh-Feldt too
rep_res$posthoc$kruskal_test
```

`compare_one_sample` *Compare one sample against a hypothesised value*

Description

Tests each feature against `mu` and returns a parametric, a rank-based and a proportion result side by side, in the same shape `compare_two_groups()` uses. There is no second group here, so the reference is a number the user supplies rather than a set of observations.

Usage

```
compare_one_sample(
  data,
  feats,
  mu = 0,
  p = 0.5,
  success = 1,
  alternative = c("two.sided", "less", "greater"),
  conf_level = 0.95,
  fc_mean = c("arith", "geom"),
  input_scale = c("raw", "log2"),
  p_adjust = "BH",
```

```
  diagnose = TRUE
)
```

Arguments

data	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
feats	Character vector of numeric column names in data to test. One output row per entry.
mu	Hypothesised value for the mean and the pseudo-median. A single number applied to every feature, read on the same scale as data, so with input_scale = "log2" it is a log2 value.
p	Hypothesised proportion for prop_test, in (0, 1).
success	The value counted as a success when a feature is binary. Defaults to 1, so a 0/1 coded column needs nothing further.
alternative	Direction of the alternative hypothesis, one of "two.sided", "less" or "greater". "greater" tests whether the sample exceeds mu, and every reported quantity follows that direction.
conf_level	Confidence level for all reported intervals.
fc_mean	Which centre the fold change divides mu into, "arith" for the arithmetic mean or "geom" for the geometric mean. Defaults to "geom" when input_scale = "log2" and to "arith" otherwise. Affects the effect table only: the t-test is a test of the mean either way.
input_scale	The scale data and mu arrive on, "raw" or "log2". On the log2 scale both are raised back through 2^x before the ratio is taken, so fold_change means what it does for raw input. This changes the effect table only, never the tests. See compare_two_groups() for the full account.
p_adjust	Multiplicity adjustment applied across feats within each test table, passed to stats::p.adjust(). Use "none" to disable.
diagnose	Logical. If TRUE, the normality check the t-test rests on is attached as $diagnostics.

Details

t_test One-sample t-test on the mean.

wilcox_test One-sample Wilcoxon signed-rank test, with the Hodges-Lehmann pseudo-median. It tests location without assuming normality, but it does assume the distribution of x - mu is symmetric.

prop_test Score test of the proportion of successes against p. It only applies to features that are binary, and features that are not come back as NA rather than being silently coerced.

Missing values are dropped per feature, so n_used varies between features when the data are incomplete. Features that cannot be tested do not abort the run: their row is filled with NA and all such features are reported together in a single warning.

Value

A sa_comparison object with the layout described in compare_two_groups(), with two differences. design carries mu, p and success instead of group_lv, since there are no groups, and effect holds n_used, center (the centre fc_mean selected), mu, diff, fold_change and log2fc, the ratio being that centre over mu. Every column of effect is on the original measurement scale, so with input_scale = "log2" its center, mu and diff are back-transformed and differ from the same-named columns of tests$t_test, which stay on the scale the tests ran on. There is no posthoc or pairwise slot: a single sample has no pair of levels to contrast.

Every test table starts with n_used and carries pval, pval_adj, lower_conf and upper_conf. The remaining columns are:

t_test center, mu, diff, stderr, t_stat, df and cohens_d, the mean difference over the sample standard deviation.

wilcox_test hl_shift, the Hodges-Lehmann pseudo-median, and v_stat.

prop_test n_success, proportion, p, diff, chi_sq, df and cohens_h. The interval is a Wilson score interval, which stays inside [0, 1] where a Wald interval does not.

Fold change against a hypothesised value

fold_change is center / mu and log2fc its base-2 logarithm, so both are undefined when mu is zero, which is also its most common value. Both columns are NA in that case rather than infinite, a message says so, and estimate_significance() will call every feature undecided. Reporting Inf would read as an infinitely large increase when what actually happened is that the question has no answer.

This cannot arise under input_scale = "log2". There the reference is 2^mu, which is positive whatever mu is, and mu = 0 simply means a reference of 1. Under the default fc_mean = "geom" the whole row reduces to log2fc = mean(x) - mu.

References

Student (1908). The probable error of a mean. *Biometrika*, 6(1), 1-25.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6), 80-83.

Wilson, E. B. (1927). Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158), 209-212.

See Also

compare_two_groups() for two groups and compare_multiple_groups() for three or more.

Examples

```
## Are the iris measurements different from 3 cm?
res <- compare_one_sample(
  data  = iris,
  feats = c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width"),
  mu    = 3
)
res
```

```
res$tests$t_test
res$tests$wilcox_test
res$effect

## A binary feature reaches the proportion test; a continuous one does not
## and comes back NA with a warning rather than being coerced.
cars <- mtcars
cars$am <- as.numeric(cars$am)
suppressWarnings(
  compare_one_sample(cars, c("am", "vs", "mpg"), mu = 0.5, p = 0.5)
)$tests$prop_test

## One-sided: is mileage above 20 mpg?
compare_one_sample(mtcars, "mpg", mu = 20,
                   alternative = "greater")$tests$t_test
```

`compare_two_groups` *Run every applicable two-group test at once*

Description

Compares exactly two group levels across any number of numeric features and returns a parametric, a rank-based and a robust test side by side, together with the fold change between the two groups. Nothing is chosen on the user's behalf: reporting all of them makes disagreement between them visible, which is the situation where the choice of test actually matters.

Usage

```
compare_two_groups(
  data,
  feats,
  group,
  group_lv,
  control_label = group_lv[1],
  id = NULL,
  alternative = c("two.sided", "less", "greater"),
  paired = FALSE,
  conf_level = 0.95,
  tr = 0.2,
  fc_mean = c("arith", "geom"),
  input_scale = c("raw", "log2"),
  p_adjust = "BH",
  diagnose = TRUE
)
```

Arguments

data	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
feats	Character vector of numeric column names in data to test. One output row per entry.
group	Grouping vector with one entry per row of data.
group_lv	Character vector of exactly two group levels. The first is the reference, treated as y, and the second is the level compared against it, treated as x, so all differences read as group_lv[2] - group_lv[1] and all ratios as group_lv[2] / group_lv[1]. Rows belonging to any other level are dropped.
control_label	The level to hold as the reference. Naming it moves that level to the front of group_lv and leaves the other one where it is, so control_label = group_lv[2] reverses every difference and ratio without the pair having to be rewritten. Defaults to group_lv[1], the reference the order already asked for.
id	Optional pairing key with one entry per row of data, used only when paired = TRUE. Supplying it matches observations by key instead of by row order, which is the safer choice whenever the rows may have been reordered or a subject may be missing from one group. Ids present in only one group are dropped.
alternative	Direction of the alternative hypothesis, one of "two.sided", "less" or "greater".
paired	Logical. If TRUE, observations are treated as matched. See id for how the pairs are formed.
conf_level	Confidence level for all reported intervals.
tr	Trimming proportion for Yuen's test, in [0, 0.5). Ignored when paired = FALSE.
fc_mean	Which centre the fold change divides, "arith" for the arithmetic mean or "geom" for the geometric mean. The geometric mean requires strictly positive values and is the usual choice for concentration-like data. Defaults to "geom" when input_scale = "log2", where it is the convention, and to "arith" otherwise.
input_scale	The scale data arrives on, "raw" or "log2". See the section below: this changes the effect table only, never the tests.
p_adjust	Multiplicity adjustment applied across feats within each test table, passed to stats::p.adjust(). Use "none" to disable.
diagnose	Logical. If TRUE, the normality and homogeneity of variance checks the tests rest on are attached as $diagnostics, computed on the same observations the tests used. See diagnose_distribution().

Details

Which member of each family is used depends on paired:

family	paired = FALSE	paired = TRUE
t	Welch's t-test	Paired t-test
Wilcoxon	Rank-sum (Mann-Whitney U)	Signed-rank
robust	Brunner-Munzel	Yuen's trimmed mean (dependent)

Direction is set once, by the order of group_lv, and every quantity in the result follows it. The first level is the reference, as it is in compare_multiple_groups(), so alternative = "greater" tests whether group_lv[2] exceeds group_lv[1] in all three families, and mean_diff, hl_shift, trim_diff, fold_change and relative_effect are all above their null value when group_lv[2] is the larger group.

control_label is the second way of stating that order. It names the level to hold as the reference and moves it to the front, so the direction can be flipped without the two levels being retyped in the other order. Because group_lv says which levels take part as well as which one is the reference, naming the second one here is a correction rather than a contradiction, which is where the comparisons differ from the models: an outcome_lv holds nothing but the two classes, so fit_logistic_regression() rejects a control_label that disagrees with it.

Features that cannot be tested do not abort the run. Their row is filled with NA and all such features are reported together in a single warning. Informational engine warnings, such as an exact p-value being unavailable because of ties, are grouped into one message().

Missing values are handled per feature: independent samples drop NA within each group, paired samples keep only complete pairs. n_x, n_y and n_used therefore vary between features when the data are incomplete. The fold change is computed from those same observations, so it never rests on a different subset of the data than the p-value beside it.

Value

A sa_comparison object: a plain list, so it survives being written out as JSON and read back in another language, with an S3 class on top that only supplies print(). Its elements are

analysis "two_group_comparison".

features The feature names, in the row order every table uses.

design group_lv, paired, pairing ("order", "id" or NA when not paired), n_dropped (rows removed for belonging to a level outside group_lv) and unmatched_ids.

parameters alternative, conf_level, tr, fc_mean, input_scale and p_adjust, as used.

effect One row per feature: x_center, y_center (the centres fc_mean selected for group_lv[2] and for the reference group_lv[1]), fold_change and log2fc.

tests t_test, wilcox_test and robust_test, described below.

test_info Per test, the method id, a readable label and whether it was the paired variant.

diagnostics Assumption checks, or NULL when diagnose is FALSE.

metadata package_version, r_version, platform and an ISO-8601 timestamp.

There is no posthoc or pairwise slot. With two groups the omnibus comparison is already the only contrast there is, so the question a post-hoc stage answers never arises. compare_multiple_groups() is where those two slots appear.

Every table in tests has one row per feature and starts with features, the per-group sample sizes n_x / n_y and n_used (total observations for independent samples, complete pairs for paired samples), and carries pval, pval_adj, lower_conf and upper_conf. Every column named for x or y reads the same way throughout: x is group_lv[2] and y is the reference group_lv[1]. The remaining columns are:

`t_test` `x_mean`, `y_mean`, `mean_diff`, `stderr`, `t_stat`, `df`. Group means are computed directly, so the columns are identical for paired and independent designs.

`wilcox_test` `hl_shift` (Hodges-Lehmann location shift, the pseudo-median of differences when paired) and `w_stat`.

`robust_test`**, independent** `relative_effect` (`P(X > Y) + 0.5 * P(X = Y)`, above 0.5 when `group_lv[2]` is the larger group), `bm_stat` and `df`. The interval is on the probability scale, so a one-sided alternative leaves it open at 0 or at 1 rather than at infinity.

`robust_test`**, paired** `x_trim_mean`, `y_trim_mean`, `trim_diff`, `stderr`, `yuen_stat`, `df` and `robust_dz`, a robust counterpart of Cohen's `dz`.

Pairing

With `paired = TRUE` and no `id`, the only available information is row order: the first row of `group_lv[1]` is matched with the first row of `group_lv[2]`, and so on. Both groups must then have the same number of rows. Note that this cannot detect rows that have been reordered, so a data set sorted differently in each group would produce wrong pairs and no complaint. Passing `id` removes that failure mode: pairs are matched on the key, ids appearing in only one group are dropped with a message, and an id repeated within a group is an error.

Log-transformed input

A ratio only means something on the scale the measurement was made on. Dividing two means of already logged values answers a different question and can even come out with the wrong sign: two log2 centres of -1 and -2 are a two-fold increase, but their ratio, 0.5, reads as a two-fold decrease. With `input_scale = "log2"` each observation is raised back through `2^x` before the centres are taken, so `fold_change` and `log2fc` mean the same thing they do for raw input and `fold_change` still equals `x_center / y_center`.

Under the default `fc_mean = "geom"` this reduces exactly to `log2fc = mean(x) - mean(y)`, the usual definition for log-scale data. `fc_mean = "arith"` is accepted but averages the back-transformed values, which the largest observations dominate, and is not the conventional log2 fold change.

Only the `effect` table is converted. The tests run on the values as supplied, which is the reason for logging them in the first place, so with `input_scale = "log2"` the centres in `effect` are on the original scale while `x_mean` and `y_mean` in `tests$t_test` are on the log2 scale. For raw input with `fc_mean = "arith"` those columns hold the same numbers; here they do not.

References

Welch, B. L. (1947). The generalization of Student's problem when several different population variances are involved. *Biometrika*, 34(1-2), 28-35.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6), 80-83.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, 18(1), 50-60.

Brunner, E. and Munzel, U. (2000). The nonparametric Behrens-Fisher problem: asymptotic theory and a small-sample approximation. *Biometrical Journal*, 42(1), 17-25.

Yuen, K. K. (1974). The two-sample trimmed t for unequal population variances. *Biometrika*, 61(1), 165-170.

See Also

estimate_significance() to reduce the result to one significance verdict per feature, draw_grouped_boxplot() to visualise the same input, and simulate_two_groups() for input whose answer is known in advance.

Examples

```
## Independent samples: two of the three iris species
iris2 <- iris[iris$Species != "setosa", ]
res <- compare_two_groups(
  data     = iris2,
  feats    = c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width"),
  group    = iris2$Species,
  group_lv = c("versicolor", "virginica")
)
res
res$tests$t_test
res$tests$robust_test
res$effect

## setosa is perfectly separated from the others on petal size, which leaves
## the Brunner-Munzel variance at zero. Those rows come back NA with a
## warning rather than aborting the other features.
suppressWarnings(
  compare_two_groups(
    data     = iris[iris$Species != "virginica", ],
    feats    = c("Sepal.Length", "Petal.Length"),
    group    = iris$Species[iris$Species != "virginica"],
    group_lv = c("setosa", "versicolor")
  )$tests$robust_test
)

## Paired samples: sleep holds 10 subjects under 2 drugs, listed in the same
## subject order within each group.
paired_res <- compare_two_groups(
  data     = sleep["extra"],
  feats    = "extra",
  group    = sleep$group,
  group_lv = c("1", "2"),
  paired   = TRUE,
  alternative = "less"
)
paired_res$tests$robust_test

## Same data with the second group shuffled. Row order pairing silently uses
## the wrong partners, while `id` recovers the correct result.
shuffled <- rbind(sleep[1:10, ], sleep[10 + c(4, 9, 1, 7, 2, 10, 3, 6, 8, 5), ])
by_order <- compare_two_groups(
  data = shuffled["extra"], feats = "extra",
  group = shuffled$group, group_lv = c("1", "2"), paired = TRUE
)
by_id <- compare_two_groups(
```

```
  data = shuffled["extra"], feats = "extra",
  group = shuffled$group, group_lv = c("1", "2"),
  id = shuffled$ID, paired = TRUE
)
## The means agree but the paired standard error, and so the p-value, do not.
rbind(order = by_order$tests$t_test,
      id    = by_id$tests$t_test)
```

`diagnose_distribution` *Check the assumptions a comparison rests on*

Description

Runs the normality tests, the homogeneity of variance tests and the outlier screen together, and reports them as one object. The same checks are attached automatically to `compare_two_groups()` and `compare_multiple_groups()`, so an assumption is never silently ignored; this function is for looking at them on their own, before a test has been chosen.

Usage

```
diagnose_distribution(
  data,
  feats,
  group = NULL,
  group_lv = NULL,
  alpha = 0.05,
  criterion = c("iqr", "robust_z", "grubbs"),
  iqr_multiplier = 1.5,
  z_threshold = 3.5,
  center = c("median", "mean", "trimmed"),
  trim = 0.1
)
```

Arguments

`data`	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
`feats`	Character vector of numeric column names in `data` to check.
`group`	Optional grouping vector with one entry per row of `data`. When supplied, normality is checked within each level and the variance tests are run across them. Without it there is only one sample per feature, so the variance table is empty.
`group_lv`	Group levels to keep, in display order. Defaults to the sorted unique values of `group`.
`alpha`	Threshold applied to the p-values when setting the `normal_ok` and `variance_ok` flags of the summary table.

`criterion`, `iqr_multiplier`, `z_threshold`
: Passed to `screen_outliers()`.

`center`, `trim`
: Centre used by the Levene test and its trimming proportion, passed through to the Brown-Forsythe variant.

Details

A failed assumption never blocks an analysis and never causes a test to be swapped for another one. It changes which member of the reported test family deserves the most weight, and that judgement stays with the user. Skewed groups make the rank-based and robust members more trustworthy than the parametric one; unequal variances make Welch's and Brunner-Munzel's treatments of the same data more trustworthy than the pooled ones.

Each assumption is checked twice on purpose, by tests that fail differently:

Normality Shapiro-Wilk is the more powerful of the two and is the one to read first. The Kolmogorov-Smirnov test is fitted against a normal with the sample's own mean and standard deviation, which makes its p-value anti-conservative, so it disagreeing with Shapiro-Wilk usually means the departure is in the tails.

Homogeneity of variance The Levene test is centred on the median, the Brown-Forsythe variant, and tolerates skew. Bartlett's test is more powerful when the groups really are normal and cannot tell unequal variances apart from heavy tails when they are not. The two disagreeing is itself evidence about normality.

Value

A `sa_diagnosis` object: a plain list carrying `analysis`, `features`, `design`, `parameters`, `metadata` and four tables.

`normality` One row per feature and group level: `features`, `group`, `n_used`, `shapiro_stat`, `shapiro_pval`, `ks_stat`, `ks_pval`, `skewness` and `excess_kurtosis`. The level column is named `group` to match `summarize_descriptive_stats()`, and is `NA` when no grouping was supplied.

`variance` One row per feature: `features`, `n_used`, `n_groups`, `levene_stat`, `levene_df1`, `levene_df2`, `levene_pval`, `bartlett_stat`, `bartlett_df` and `bartlett_pval`. Zero rows when no `group` was supplied.

`outliers` The `screen_outliers()` table, one row per flagged observation.

`summary` One row per feature: `features`, `n_levels`, `n_outliers`, `min_shapiro_pval`, `normal_ok` and `variance_ok`. The two flags are `NA` when the corresponding test could not be run.

The result is deliberately not an `sa_comparison`. Normality is a property of one sample and homogeneity a property of a set of them, so the two tables have different numbers of rows and would not fit a contract built around one row per feature.

References

Shapiro, S. S. and Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4), 591-611.

Massey, F. J. (1951). The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253), 68-78.

Brown, M. B. and Forsythe, A. B. (1974). Robust tests for the equality of variances. *Journal of the American Statistical Association*, 69(346), 364-367.

Bartlett, M. S. (1937). Properties of sufficiency and statistical tests. *Proceedings of the Royal Society A*, 160(901), 268-282.

See Also

`screen_outliers()` for the outlier stage on its own, and `compare_multiple_groups()`, whose `$diagnostics` slot holds the same tables for the data it tested.

Examples

```
feats <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")

## Within species, so a genuine species difference is not read as skew
d <- diagnose_distribution(iris, feats, iris$Species)
d
d$normality
d$variance
d$summary

## Petal length pooled over species is strongly bimodal, which every
## normality test picks up once the grouping is taken away.
diagnose_distribution(iris, "Petal.Length")$normality
```

---

draw_butterfly_hist *Draw a butterfly histogram of one feature across two groups*

Description

Draws the distribution of a single feature for two group levels back to back, on a shared set of breaks. The first level runs left from the centre line and the second one right, so the two shapes can be compared bin by bin instead of read off two separate panels.

Usage

```
draw_butterfly_hist(
  data,
  feat,
  group,
  group_lv,
  breaks = "Sturges",
  scale = c("count", "proportion", "density"),
  type = c("freq", "dens", "both"),
  dens_adjust = 1,
  dens_lwd = 2,
  dens_col = NULL,
```

```
  dens_alpha = 0.45,
  col = c("#4575B4", "#D73027"),
  border = "white",
  xlab = NULL,
  ylab = NULL,
  main = NULL,
  cex.lab = 1.3,
  cex.axis = 1.2,
  cex.main = 1.3,
  cex.legend = 1.1,
  legend.position = "topright",
  xlim = NULL,
  ylim = NULL,
  margin = c(5, 5, 4, 3),
  out_statistics = TRUE,
  ...
)
```

Arguments

data	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
feat	Name of the numeric column in data to plot. One feature per call.
group	Grouping vector with one entry per row of data.
group_lv	Character vector of exactly two group levels. The first is drawn on the left, the second on the right. Rows belonging to any other level are dropped.
breaks	Histogram break specification, shared by both groups. Accepts a character rule supported by graphics::hist(), a single number read as the approximate bin count, or a strictly increasing vector of break points.
scale	What the bar length means: "count", "proportion" of the group, or "density". Proportions and densities are computed within each group, so unequal group sizes still give comparable shapes. When type is not "freq" this defaults to "density", and any other value given explicitly is an error, since only the density scale puts the curve and the bars on one axis.
type	Which layers to draw: "freq" for the histogram bars alone, "dens" for the kernel density estimate alone, drawn as a filled shape, or "both" for the two overlaid, the density shape on top of the bars.
dens_adjust	Bandwidth multiplier passed to stats::density() as adjust. Values above 1 smooth the shape further.
dens_lwd	Line width of the density outline.
dens_col	Colour of the density outline, either one colour or one per group level. NULL outlines each group in its own col.
dens_alpha	Opacity of the density fill, between 0 and 1. The default leaves the shape half transparent, so under type = "both" the bars stay visible through it and the two layers read as one distribution rather than two. Needs a device that supports transparency; set it to 1 if yours does not, and use type = "dens" there.

col	Length-2 vector of fill colours, for the left and right group.
border	Colour of the bar borders.
xlab, ylab, main	Axis labels and title. xlab defaults to the meaning of scale and ylab to feat.
cex.lab, cex.axis, cex.main, cex.legend	Character expansion for the axis labels, the axis annotation, the title and the legend.
legend.position	Position passed to graphics::legend(), or NULL / FALSE to leave the legend out.
xlim, ylim	Numeric length-2 ranges for the bar axis and the value axis, or NULL to derive them from the data. xlim is not forced to be symmetric when it is supplied. A derived ylim covers the breaks, and the tails of the density estimate as well when one is drawn.
margin	Plot margins in lines, passed to graphics::par() as mar.
out_statistics	If TRUE, invisibly return the numbers behind the bars.
...	Additional arguments passed to graphics::plot.default().

Details

The left group is drawn at negative coordinates so that both distributions share one axis, but the tick labels are the absolute values, so a bar is read the same way on either side.

A density curve and a bar can only be read against the same axis when the bar is a density too: a count or a proportion per bin scales with the bin width, which the curve knows nothing about. So type = "dens" and type = "both" move scale to "density", and reject a scale that was asked for explicitly and says otherwise, rather than drawing two incomparable shapes.

The plot starts from the device background rather than imposing a theme of its own, as draw_volcano_plot() does. Only two colours are needed here, so there is no dark variant; that is reserved for draw_grouped_boxplot(), where three or more group colours have to stay apart.

Only the margins are restored on exit, not every graphical parameter, so the coordinate system survives the call and graphics::abline(), graphics::points() and friends can still add to the finished plot.

Value

If out_statistics = FALSE, NULL invisibly. Otherwise a list, invisibly:

bin_summary_stats One row per bin, with bin_start, bin_end, bin_mid and one column per group level holding the plotted bar length. What that length means follows scale.

group_summary_stats One column per group level with rows n (finite values used), n_dropped (missing or non-finite values left out), min and max.

group_hists One "histogram" object per group level, named by the level, exactly as graphics::hist() returns them, so plot(res$group_hists[[lv]]) redraws that group on its own. Both groups are binned on the shared breaks, so these are not what calling graphics::hist() on one group alone would give: that would pick its own breaks from that group only.

group_densities Present only when type is not "freq". One "density" object per group level, named by the level, as stats::density() returns them. The bins are still reported in bin_summary_stats even when no bars were drawn.

See Also

draw_grouped_boxplot() for several features at once, and compare_two_groups() to test the difference the plot shows.

Examples

```
iris2 <- iris[iris$Species != "setosa", ]

draw_butterfly_hist(
  data     = iris2,
  feat     = "Petal.Length",
  group    = iris2$Species,
  group_lv = c("versicolor", "virginica"),
  breaks   = 12,
  scale    = "proportion",
  main     = "virginica vs versicolor"
)

## Counts, plus the numbers behind the bars
res <- draw_butterfly_hist(iris2, "Sepal.Width", iris2$Species,
                           c("versicolor", "virginica"))
res$bin_summary_stats
res$group_summary_stats

## Each group also comes back as a plain histogram object
plot(res$group_hists$virginica)

## Bars with a kernel density estimate on top, both on the density scale
res <- draw_butterfly_hist(
  data     = iris2,
  feat     = "Petal.Length",
  group    = iris2$Species,
  group_lv = c("versicolor", "virginica"),
  breaks   = 12,
  type     = "both",
  main     = "Petal length, bars and density"
)
res$group_densities$virginica

## The smoothed shapes on their own
draw_butterfly_hist(iris2, "Petal.Length", iris2$Species,
                    c("versicolor", "virginica"), type = "dens")
```

`draw_corrplot` *Draw a correlation matrix, with the cells that failed the test left blank*

Description

A corrplot: one cell per pair of features, coloured by the coefficient on a fixed -1 to 1 scale, with both axes in one order so that the diagonal runs corner to corner and blocks of features that move together sit next to each other. Given the p-values as well, the pairs that did not clear `sig_level` are drawn as blank cells, so that what is left coloured is what there is evidence for.

Usage

```
draw_corrplot(
  cor_matrix,
  method = NULL,
  pvalue = NULL,
  use_adjusted = TRUE,
  sig_level = 0.05,
  cluster = TRUE,
  hclust_method = c("average", "complete", "ward.D2"),
  zlim = c(-1, 1),
  anno = TRUE,
  main = NULL,
  cex.anno = 1,
  cex.axis = 0.9,
  cex.main = 1.5,
  cex.legend = 1.2,
  ...
)
```

Arguments

`cor_matrix`	The result of `summarize_association_stats()`, or a correlation matrix on its own. A matrix must be square and symmetric, with the features as its dimnames, and hold at least two of them.
`method`	Which coefficient to draw when `cor_matrix` is a result, naming one of the slots it holds. `NULL` takes the first method it was computed with. It must be `NULL` when `cor_matrix` is a matrix, there being only one matrix to draw.
`pvalue`	Matrix of p-values laid out like `cor_matrix`, used only when `cor_matrix` is a matrix. When `cor_matrix` is a result the p-values come from the same slot as the coefficients, so this must be left `NULL`.
`use_adjusted`	Whether to read `adj_pvalue` rather than `pvalue` out of the result. Ignored when `pvalue` is supplied directly.
`sig_level`	Largest p-value a cell may have and still be drawn. Cells above it are blanked. The diagonal is never blanked, a feature not being tested against itself.
`cluster`	Whether to reorder the features by clustering them. `FALSE` keeps the order they arrive in.
`hclust_method`	Linkage handed to `stats::hclust()`.
`zlim`	Numeric length-2 range the colours span. The default is the range a correlation can take, so that the same colour means the same strength from one plot to the next.

anno — Whether to write each coefficient on its cell, rounded to two decimal places by draw_heatmap(). Blanked cells stay blank.

main — Plot title.

cex.anno — Character expansion for those cell labels, relative to the smaller of the feature axis label sizes.

cex.axis, cex.main, cex.legend — Character expansion for the axis labels, title and colour key, passed to draw_heatmap().

... — Passed to draw_heatmap(). The arguments this function decides (data, group, group_lv, scale, cluster_feats, cluster_samples, anno, cex.anno, cex.axis, cex.main, cex.legend) cannot be given again.

Details

draw_heatmap() draws it. This function decides what it is handed: the matrix unscaled, the colour range fixed at the range a correlation can take, one clustering shared by the two axes, and the blanking applied afterwards.

The distance the clustering runs on is 1 - cor(), the same one cluster_hclust() and draw_heatmap() mean by dist_method = "correlation", so a corrplot and a heatmap of the same features group them the same way. It is computed once and both axes are permuted by it. A matrix holding an NA, which is what a feature with no variance leaves behind, has no distance for that feature, and rather than fail the features are left in their input order with a message saying so.

Blanking happens after the clustering rather than before it. A cell removed for its p-value would otherwise change the tree, and the picture would no longer be the matrix the reader is being shown.

A cell whose p-value is NA, a pair that could not be tested, is left as it arrived rather than blanked: there is no evidence against it either, and the coefficient beside it is usually already NA.

The matrix is symmetric, so the transpose draw_heatmap() takes on the way in leaves it unchanged and the features come out on both axes.

Value

A list, invisibly: everything draw_heatmap() returns, and beside it

corr The matrix as it was drawn, in the drawn order and with the blanked cells NA.

pvalue The p-values in that same order, or NULL.

order The permutation of the input the clustering chose.

hclust The stats::hclust() object behind it, or NULL when the features were not clustered.

n_masked How many cells were blanked.

See Also

summarize_association_stats(), which produces what this draws, and draw_heatmap(), which draws it.

Examples

```
feats <- c("mpg", "cyl", "disp", "hp", "drat", "wt")
res <- summarize_association_stats(mtcars, feats, methods = "pearson")

## Every pair, with the features clustered so that the blocks sit together
drawn <- draw_corrplot(res, main = "mtcars")
drawn$order

## Only the pairs that cleared the adjustment at 1%
draw_corrplot(res, sig_level = 0.01, cex.anno = 0.8, cex.axis = 0.7)

## A bare matrix, in the order it arrives and with nothing to blank
draw_corrplot(stats::cor(mtcars[feats]), cluster = FALSE)
```

`draw_dim_reduction_plot` *Draw a reduction as a scatter of its points*

Description

Plots two coordinates of a `perform_pca()`, `perform_tsne()` or `perform_umap()` result against each other. A clustering of the same frame colours the points and a known grouping shapes them, so what the data was found to say and what the caller already knew are read off one picture.

Usage

```
draw_dim_reduction_plot(
  reduction_result,
  group = NULL,
  group_lv = NULL,
  cluster_result = NULL,
  cluster_lv = NULL,
  dims = c(1L, 2L),
  anno_points = FALSE,
  dark = FALSE,
  asp = NULL,
  col = NULL,
  pch = NULL,
  cex = 1.2,
  xlim = NULL,
  ylim = NULL,
  xlab = NULL,
  ylab = NULL,
  main = NULL,
  cex.axis = 1.2,
  cex.lab = 1.3,
```

```
  cex.main = 1.3,
  cex.legend = 1.1,
  cex.anno = NULL
)

## S3 method for class 'sa_reduction'
plot(x, ...)
```

Arguments

reduction_result	A reduction, as returned by perform_pca(), perform_tsne() or perform_umap().
group	One label per point, or NULL for no shape. Points are the rows the reduction kept, which reduction_result$points names, and on the feature scale they are features rather than samples.
group_lv	The levels of group in the order they are drawn and listed in, or NULL for the factor's own order and otherwise alphabetical. Unlike draw_grouped_boxplot()'s, this argument selects no rows: a level group uses and group_lv leaves out is an error rather than a point dropped from the picture.
cluster_result	A clustering of the same points, as returned by cluster_hclust(), cluster_kmeans(), cluster_dbscan() or cluster_snn(), or NULL for no colouring.
cluster_lv	One label per cluster, in the order the clustering numbers them, or NULL for #1, #2, and so on. Noise is still listed as noise (n) when present.
dims	Which two coordinates to draw, as positions in the score table.
anno_points	Whether to write each point's label beside it.
dark	Whether to draw on a dark background.
asp	Aspect ratio passed to graphics::plot(), or NULL to let the axes fill the panel independently. 1 makes distances comparable between the two axes.
col	One colour for every point, one per cluster when cluster_result is given, one per group level when only group is given, or NULL for hcl.colors(n, "Dark 2") on clusters and the foreground colour elsewhere. Noise is grey whatever this says.
pch	One shape for every point, one per group level when group is given, or NULL for 16 and the default shape sequence respectively.
cex	Size of the plotted points.
xlim, ylim	Axis ranges, or NULL to take them from the coordinates.
xlab, ylab, main	Axis and figure labels. NULL builds them from the reduction.
cex.axis, cex.lab, cex.main, cex.legend, cex.anno	Relative text sizes. cex.anno sizes the point labels and NULL matches cex.legend.
x	A reduction, as returned by perform_pca(), perform_tsne() or perform_umap().
...	Arguments passed on to draw_dim_reduction_plot().

Value

A data.frame of the points as they were drawn, invisibly: `points`, `x`, `y`, `col`, `pch`, and `cluster` and `group` when those were given. The resolved colouring is carried as a `"view"` attribute, one of `"both"`, `"cluster"`, `"group"` or `"plain"`.

Colour and shape say two different things

`cluster_result` takes the colours and `group` takes the shapes when both are given, and giving both is the point of the arrangement rather than a conflict to be resolved. With one channel alone, that channel takes the colours when `col` is named; otherwise a lone clustering is coloured and a lone grouping is shaped. The question a clustering usually raises is whether it recovered a grouping that was known all along, and on two channels that is read directly: one colour per shape is a clustering that found the groups, and a shape split across colours is a group the data does not see as one thing. Give neither and the points are drawn in the foreground colour.

The clustering has to be of the same points, which the two contracts already promise: `sa_cluster` and `sa_reduction` both read their input through the same function, so a mismatch is refused rather than lined up by position.

Noise

`cluster_dbscan()` and `cluster_snn()` can leave a point in no cluster at all. Those points are grey rather than a palette colour, since a point left out is the absence of a cluster and not a cluster of its own, and the legend counts them on a line of their own.

The axes

A principal component analysis reports what share of the variance each of its components carries, so its axis labels carry it too: `PC1 (30.2%)` is read from `$variance` rather than recomputed. An embedding has no such number and its axes are labelled with their names alone. `asp = 1` is what makes one unit of the vertical axis the same length as one unit of the horizontal, which is worth setting when the distance between two points is what is being read.

See Also

`perform_pca()` for the coordinates and `cluster_kmeans()` for the labels, and `draw_heatmap()`, which shows the same wide input a cell at a time rather than a point at a time.

Examples

```
res <- perform_pca(iris[1:4])

## What was known all along, as shapes. Name `col` to colour the levels
## instead.
draw_dim_reduction_plot(res, group = iris$Species)

draw_dim_reduction_plot(res, group = iris$Species,
                        col = c("#E69F00", "#56B4E9", "#009E73"))

## What the data was found to say, as colours. The clustering was never shown
## the species, so one colour per shape is its own finding.
```

```
cl <- cluster_kmeans(iris[1:4], n_clust = 3, seed = 1)
drawn <- draw_dim_reduction_plot(res, group = iris$Species,
                                 cluster_result = cl)
table(cluster = drawn$cluster, species = drawn$group)

## Names for the clusters the legend would otherwise number.
draw_dim_reduction_plot(res, cluster_result = cl,
                        cluster_lv = c("A", "B", "C"))

## A density method can place no point at all, and those are grey.
db <- cluster_dbscan(iris[1:4])
draw_dim_reduction_plot(res, cluster_result = db, asp = 1)

## The features as the points, labelled, on the third and fourth components.
by_feat <- perform_pca(iris[1:4], embedding_scale = "features")
draw_dim_reduction_plot(by_feat, dims = c(1, 3), anno_points = TRUE)
```

`draw_forest_plot` *Draw a forest plot of a comparison result*

Description

Draws the estimate of each feature beside its confidence interval, falling back to a bar of `-log10(pval_adj)` for a table that has no interval to draw, which is every omnibus test.

Usage

```
draw_forest_plot(
  comparison_result,
  test = names(comparison_result$tests)[1],
  type = c("auto", "estimate", "posthoc", "pvalue"),
  feats = NULL,
  use_adjusted = TRUE,
  alpha = 0.05,
  sort_by = c("none", "pvalue"),
  dark = FALSE,
  xlim = NULL,
  xlab = NULL,
  main = NULL,
  col_signif = "#D1495B",
  col_plain = "#7F8C8D",
  cex.axis = 0.9,
  cex.lab = 1.1,
  cex.main = 1.2,
  cex.legend = 1,
  ...
)
```

```
## S3 method for class 'sa_comparison'
plot(x, ...)
```

Arguments

comparison_result	A comparison result, as returned by compare_two_groups(), compare_multiple_groups() or compare_one_sample().
test	Which test to draw. One of names(comparison_result$tests).
type	"auto", "estimate", "posthoc" or "pvalue".
feats	Character vector of features to draw, in display order from the top of the plot down. NULL, the default, draws every feature of the chosen table, except in the post-hoc view, where it draws the first feature of the post-hoc table. A feature that has no contrasts because its omnibus test did not qualify is reported in a message() and left out.
use_adjusted	Logical. If TRUE, read the pval_adj column; if FALSE, the unadjusted pval. The colouring, the sorting, the p-value view and the labels all follow, so the plot always describes the p-value it actually used.
alpha	Threshold marked on the p-value view and used to colour the points of the estimate view.
sort_by	"none" to keep the feature order of the result, or "pvalue" to draw the most significant rows at the top.
dark	If TRUE, use a dark background with light text.
xlim	Numeric length-2 x axis range, or NULL to derive it from the values being drawn. A supplied range is used as given, and the interval bounds are clamped to it, so an interval running past the range still reaches the edge of the panel instead of vanishing.
xlab, main	Axis and title labels. Both are derived from the result when left NULL.
col_signif, col_plain	Colours for rows at or below alpha and for the rest.
cex.axis, cex.lab, cex.main, cex.legend	Character expansion for the axis annotation, the axis label, the title and the legend.
...	Arguments the plot() method passes on. Anything draw_forest_plot() does not name is ignored.
x	The same comparison result, under the name the plot() generic fixes for its first argument.

Details

Only the columns the result contract guarantees are read, which is why one function covers every scenario: it never asks whether the object came from two groups, three groups or a single sample, only whether the table it was handed has intervals or p-values, and every table has one or the other. plot() on an sa_comparison is the same function under the name R users reach for first.

Three views are available:

`"estimate"` A forest plot of the effect estimate and its confidence interval, one row per feature. Available whenever the chosen table has finite intervals, which the two-group and one-sample scenarios always do.

`"posthoc"` The same forest plot for the pairwise contrasts of a multi-group comparison, one row per contrast. Only the first feature of the post-hoc table is drawn unless `feats` names the ones to draw. An estimate reads as `group1 - group2`, the direction the row label spells out, and the reference level is the one subtracted, so a point to the right of the guide agrees in sign with the `log2fc` a volcano plot of the same comparison draws.

`"pvalue"` `-log10()` of the p-value per feature with the `alpha` threshold marked. The fallback when a table has no interval to draw, which is the case for every omnibus test.

`type = "auto"`, the default, picks the first of those three that the chosen table can actually support.

The function changes graphical parameters and the panel layout, and restores both on exit, so the caller's device is left as it was found. The legend lives in a narrow panel of its own on the right, where it cannot cover the rows it describes.

The estimate view marks the null value with a vertical line, at zero for a difference and at one for the Brunner-Munzel relative effect, which is the one quantity in the package whose null is not zero.

Value

The plotted data.frame, invisibly, in the row order it was drawn, with the view that `type = "auto"` resolved to attached as the attribute `"view"`.

See Also

`draw_volcano_plot()`, which plots effect size against significance rather than estimate against interval, and `draw_grouped_boxplot()` for the input data.

Examples

```
iris2 <- iris[iris$Species != "setosa", ]
feats <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")
res <- compare_two_groups(iris2, feats, iris2$Species,
                          c("virginica", "versicolor"))

## Mean differences with their intervals
draw_forest_plot(res)

## plot() reaches the same function
plot(res)

## The same comparison read through the rank-based test
draw_forest_plot(res, test = "wilcox_test", sort_by = "pvalue")

## Two features only, drawn in the order they are named, against the
## unadjusted p-value
draw_forest_plot(res, feats = c("Petal.Width", "Petal.Length"),
                 use_adjusted = FALSE)

## An axis range of your own, so that two plots can be read against each other
```

```
draw_forest_plot(res, xlim = c(0, 2))

## A multi-group omnibus table has no interval, so "auto" falls through to
## the pairwise contrasts.
multi <- compare_multiple_groups(iris, feats, iris$Species,
                                 levels(iris$Species))
draw_forest_plot(multi, feats = "Petal.Length")

## Several features at once label each contrast with the feature it belongs to
draw_forest_plot(multi, feats = c("Petal.Length", "Sepal.Width"))
draw_forest_plot(multi, type = "pvalue")
```

`draw_grouped_barplot` *Draw a grouped barplot of a descriptive summary*

Description

Draws one bar per feature and group level, the levels side by side inside each feature's cluster and a legend in a narrow panel on the right. The bar heights are one column of `summarize_descriptive_stats()`, read from the same wide input the comparison functions take, so a bar and a row of that table are the same number and neither has to be recomputed to check the other.

Usage

```
draw_grouped_barplot(
  data,
  feats,
  group = NULL,
  group_lv = NULL,
  control_label = NULL,
  mainbar = c("mean", "median", "n", "n_missing", "sd", "var", "se", "cv", "mad",
    "skewness", "excess_kurtosis"),
  errorbar = c("none", "se", "sd", "ci"),
  conf_level = 0.95,
  gap = 1,
  lwd = 1.5,
  col = NULL,
  xlab = NULL,
  ylab = NULL,
  main = NULL,
  ylim = NULL,
  dark = FALSE,
  grid_lty = 1,
  grid_lwd = 0.25,
  cex.lab = 1.3,
  cex.axis = 1.2,
  cex.main = 1.3,
```

```
  cex.legend = 1.1,
  out_statistics = TRUE
)
```

Arguments

data	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
feats	Character vector of numeric column names in data to plot, in display order along the x axis.
group	Grouping vector with one entry per row of data. Required: a summary of every row together is what summarize_descriptive_stats() returns without one, and it has no clusters to draw.
group_lv	Character vector of at least two group levels, in the order they should appear inside each cluster. Rows belonging to any other level are dropped. NULL takes the levels from group the way summarize_descriptive_stats() does: the factor levels if group is a factor, the sorted unique values otherwise.
control_label	The level to hold as the reference. It moves to the front of group_lv and the rest keep the order they were given, the same move compare_two_groups() and compare_multiple_groups() make when the same argument is passed, so a figure and the analysis it illustrates put the reference bar in the same place when they are given the same argument. Read only when group_lv is supplied or derived: it re-points a list that already exists rather than stating the reference for the first time.
mainbar	Which column of summarize_descriptive_stats() the bar heights are. See "What a bar may carry an interval for" for which of them also take an errorbar.
errorbar	What the bars either side of a height are. "none", "se" for one standard error, "sd" for one standard deviation, or "ci" for an interval at conf_level. Read under mainbar, which decides which of them the height has an answer for.
conf_level	Confidence level of errorbar = "ci", read only under mainbar = "mean". The median's interval is the notch, whose width is fixed at the 1.58 that makes it an approximate 95% interval, so there is no level for it to be stated at.
gap	Blank bar widths inserted between neighbouring clusters.
lwd	Line width of the error bars and of the baseline when the heights run both ways.
col	Fill colours for the group levels, recycled if short. NULL takes them from grDevices::hcl.colors(), the palette draw_grouped_boxplot() and draw_interaction_plot() already draw group levels in.
xlab, ylab, main	Axis and title labels. ylab left NULL names mainbar, since the axis of a bar chart is a particular statistic rather than the measurement itself; pass "" for no label at all.
ylim	Numeric length-2 y axis range, or NULL to derive one that covers the bars, their intervals and the zero they are measured from.
dark	If TRUE, use a dark background with light text.
grid_lty, grid_lwd	Line type and width of the horizontal grid.

`cex.lab`, `cex.axis`, `cex.main`, `cex.legend`
: Character expansion for axis labels, axis annotation, the main title and the legend.

`out_statistics`
: If `TRUE`, invisibly return the numbers behind the bars.

Details

This is the summary counterpart of `draw_grouped_boxplot()`. A box shows the distribution a group's observations have; a bar shows one number standing for them, which is less of the data and is what a figure wants when the point being made is about a location rather than a spread.

A bar is read against the zero it stands on, so a derived `ylim` always includes zero and puts its headroom on the side the bars run to. Heights that go both ways, which `"skewness"` and a `"mean"` of centred features do, get the baseline drawn as well.

A bar whose height is `NA` leaves a blank rather than shifting the ones beside it, which is what makes a group too small for a shape estimate visible instead of silently absent.

The function changes graphical parameters and the panel layout, and restores both on exit, so the caller's device is left as it was found.

Value

If `out_statistics = FALSE`, `NULL` invisibly. Otherwise, invisibly, a data.frame of one row per bar in the order they were drawn, which is the row order of `summarize_descriptive_stats()`: a feature's levels stay together, in `group_lv` order.

`features`,`group` Which bar the row is.

`n` Finite observations the bar was computed from.

`value` The bar height, the `mainbar` column.

`lower`,`upper` The ends of the interval, `NA` under `errorbar = "none"` and for a bar whose interval was not defined.

Which height and which interval were drawn are attached as the attributes `"mainbar"` and `"errorbar"`.

What a bar may carry an interval for

`mainbar` names any of the summary columns, but only two of them are locations that an interval either side says something about, so `errorbar` is read under `mainbar` rather than independently of it.

`"mean"` Takes every bar. `"se"` and `"sd"` are one standard error and one standard deviation either side, and `"ci"` is Student's interval at `conf_level`: `stats::qt()` on `n - 1` degrees of freedom times the standard error.

`"median"` Takes `"ci"` only, the notch interval `median +/- 1.58 * IQR / sqrt(n)`. That is the same interval `draw_grouped_boxplot()` returns as `median_confidence_stats`, so a bar and the notch of the box beside it are the same width on the same data. `"se"` and `"sd"` describe the observations' spread about their mean and are refused here rather than drawn around a median they are not about.

Everything else Takes `"none"`. A height that is itself a spread, a count or a shape has no second quantity for an interval to be about.

A refused combination is an error rather than a silently dropped interval, because the alternative is a figure that answers a question other than the one it was asked.

References

McGill, R., Tukey, J. W. and Larsen, W. A. (1978). Variations of box plots. *The American Statistician*, 32(1), 12-16.

See Also

`summarize_descriptive_stats()` for the table the heights come from, `draw_grouped_boxplot()` for the observations behind them, and `compare_two_groups()` or `compare_multiple_groups()` to test the difference a pair of bars shows.

Examples

```
feats <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")

## Mean per species, one standard error either side
bars <- draw_grouped_barplot(iris, feats, iris$Species, errorbar = "se")
head(bars)

## The heights are the table's own column, so neither has to be checked
## against the other by eye.
summ <- summarize_descriptive_stats(iris, feats, iris$Species)
all.equal(bars$value, summ$mean)

## Student's interval, at a level of your own
draw_grouped_barplot(iris, feats, iris$Species, errorbar = "ci",
                     conf_level = 0.99)

## A median takes the notch interval draw_grouped_boxplot() notches with
draw_grouped_barplot(iris, feats, iris$Species, mainbar = "median",
                     errorbar = "ci")

## A count carries no interval, and `group_lv` picks the levels and their
## order
draw_grouped_barplot(iris, feats, iris$Species,
                     group_lv = c("virginica", "versicolor"),
                     mainbar = "n")

## `control_label` draws the named level first without rewriting `group_lv`
draw_grouped_barplot(iris, feats, iris$Species,
                     control_label = "setosa")
```

`draw_grouped_boxplot` *Draw a grouped boxplot across several features*

Description

Draws clusters of boxes with the levels of one factor side by side inside each cluster and a legend in a narrow panel on the right. A single factor gives one cluster per feature. A crossed design gives one panel per feature, with the factors after the first along the x axis, so that the crossing sits inside a panel where an interaction can be seen. Optionally returns the summary statistics behind the boxes.

Usage

```
draw_grouped_boxplot(
  data,
  feats,
  group = NULL,
  group_lv = NULL,
  factors = NULL,
  factor_lv = NULL,
  control_label = NULL,
  panel_by = c("feature", "factor"),
  panel_nrow = NULL,
  gap = 1,
  lwd = 1.5,
  xlab = NULL,
  ylab = NULL,
  cex.lab = 1.3,
  cex.axis = 1.2,
  cex.main = 1.3,
  ylim = NULL,
  main = NULL,
  dark = FALSE,
  grid_lty = 1,
  grid_lwd = 0.25,
  cex.legend = 1.1,
  out_statistics = TRUE
)
```

Arguments

`data`	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
`feats`	Character vector of numeric column names in `data` to plot, in display order along the x axis.
`group`	Grouping vector with one entry per row of `data`, for a single factor. Leave it `NULL` and name `factors` to draw a crossed design.

`group_lv` Character vector of at least two group levels, in the order they should appear inside each cluster. Rows belonging to any other level are dropped.

`factors` Named list of the crossed factors, each entry either the name of a column of `data` or a vector with one entry per row of it, exactly as `compare_factorial_groups()` takes them. There have to be at least two: one factor is `group`. The **first factor is the primary one**, the factor whose levels are the coloured boxes inside a cluster and the entries of the legend; the factors after it are crossed into the other axis.

`factor_lv` Named list giving the levels of each factor, with the reference level first, or `NULL` to take the levels from the data in sorted order. Naming it fixes the order the boxes, the clusters and the panels are drawn in and drops the rows belonging to any level it leaves out.

`control_label` The level to hold as the reference, one name per factor it points at, as a named list (`list(treatment = "control", sex = "male")`) or as a named character vector, exactly as `compare_factorial_groups()` takes it. The level it names is drawn first in its own factor, so a figure and the analysis it illustrates put the reference cell in the same place when they are given the same argument. Read only under `factors`: a single factor draws in `group_lv` order, which already says where its reference is.

`panel_by` Which axis the panels are over, read only under `factors`: `"feature"` for one panel per feature with the remaining factors along the x axis, or `"factor"` for one panel per combination of the remaining factors with the features along the x axis. See "What is drawn where" for why the first is the default.

`panel_nrow` How many rows the panels are laid out in, or `NULL` to let the arrangement decide: one row under `panel_by = "factor"`, and a grid as near square as the panel count allows under `"feature"`, where there is one panel per feature and a row of ten of them cannot be read.

`gap` Blank box widths inserted between neighbouring clusters.

`lwd` Line width of the boxes.

`xlab`, `ylab`, `main` Axis and title labels.

`cex.lab`, `cex.axis`, `cex.main`, `cex.legend` Character expansion for axis labels, axis annotation, the main title and the legend.

`ylim` Numeric length-2 y axis range, shared by every panel. `NULL` derives it, and what it derives depends on what a panel is. Panels over the factors hold the same features and share one range taken from every value drawn, since panels of the same quantity whose axes differ cannot be read against each other. Panels over the features hold different quantities, each with its own baseline, so each keeps the range `graphics::boxplot()` gives it and is annotated with its own axis.

`dark` If `TRUE`, use a dark background with light text.

`grid_lty`, `grid_lwd` Line type and width of the horizontal grid.

`out_statistics` If `TRUE`, invisibly return the summary statistics used by the plot.

Details

The function changes graphical parameters and the panel layout, and restores both on exit, so the caller's device is left as it was found.

Value

If `out_statistics = FALSE`, `NULL` invisibly. Otherwise a list of two elements, invisibly:

`box_summary_stats` One data.frame per feature with rows `min`, `lower_bound`, `Q1`, `median`, `Q3`, `upper_bound`, `max` and one column per box: the group levels under `group`, and the cell labels of the design under `factors`. The bounds are the Tukey whisker fences `Q1 - 1.5 * IQR` and `Q3 + 1.5 * IQR`, not the drawn whisker ends.

`median_confidence_stats` One data.frame per feature with rows `n`, `lower_conf`, `upper_conf`, the notch interval `median +/- 1.58 * IQR / sqrt(n)`. `n` counts non-missing values.

What is drawn where

A crossed design has three categorical axes to place and two dimensions to place them in, so one of them has to become the panels. Which one decides whether the picture shows an interaction.

`panel_by = "feature"`, the default, gives one panel per feature. Inside it the remaining factors run along the x axis and the primary factor is the coloured boxes within each of those clusters, so **the two factors are side by side in one panel**: an effect of the treatment that reverses between the levels of the other factor is a pattern of colours that visibly flips a couple of centimetres away, which is what an interaction is.

`panel_by = "factor"` gives the transpose: one panel per combination of the remaining factors, with the features along the x axis. It puts every feature of one cell together, which is what to ask for when the question is about the features rather than about the crossing. The cost is that the two factors are then split between the legend and the panels, and reading an interaction means comparing a colour profile in one panel against the same in another.

Either way the boxes are the same boxes and the returned statistics are identical; only the grouping of them into panels and clusters differs. The cells, their order and their labels come from the same helpers `compare_factorial_groups()` uses, so a box of this plot and a row of that result are the same observations, and the columns of the returned statistics are the cell labels the comparison and `simulate_factorial_groups()` both key on. A cell holding no observation leaves its box blank rather than shifting the ones beside it, and is reported in a message.

References

McGill, R., Tukey, J. W. and Larsen, W. A. (1978). Variations of box plots. *The American Statistician*, 32(1), 12-16.

See Also

`compare_two_groups()` and `compare_factorial_groups()` to test the same input.

Examples

```
## All three iris species across the four measurements
stats <- draw_grouped_boxplot(
  data     = iris,
  feats    = c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width"),
  group    = iris$Species,
  group_lv = c("setosa", "versicolor", "virginica"),
```

```
  ylab     = "cm",
  dark     = FALSE
)
stats$box_summary_stats$Sepal.Length
stats$median_confidence_stats$Sepal.Length

## A crossed design: one panel per feature, the sexes along the x axis and
## the treatment the colours, so an effect that flips with sex is local.
sim <- simulate_factorial_groups(n_feats = 6, n_per_cell = 8, seed = 1)
cells <- draw_grouped_boxplot(
  data      = sim$args$data,
  feats     = sim$args$feats,
  factors   = sim$args$factors,
  factor_lv = sim$args$factor_lv,
  ylab      = "log2 abundance"
)
## One column per cell, labelled the way the answer key labels them.
cells$box_summary_stats$prot_1

## The transpose: one panel per sex, every feature together inside it.
draw_grouped_boxplot(
  data      = sim$args$data,
  feats     = sim$args$feats,
  factors   = sim$args$factors,
  factor_lv = sim$args$factor_lv,
  panel_by  = "factor",
  ylab      = "log2 abundance"
)

## `control_label` draws the named level first in its own factor, and takes
## the same argument the comparison does, so the two agree on where the
## reference cell sits.
draw_grouped_boxplot(
  data          = sim$args$data,
  feats         = sim$args$feats[1:2],
  factors       = sim$args$factors,
  control_label = list(sex = "female"),
  ylab          = "log2 abundance"
)
```

`draw_heatmap` *Draw a clustered heatmap of features by samples*

Description

Draws one cell per feature and sample, with the sample groups as a coloured annotation strip above the columns and a dendrogram on each axis that was clustered. The input is the wide format the comparison functions take, one row per observation and one column per feature, and it is transposed so that features run down the rows the way an expression heatmap is usually read.

stats::heatmap() draws the cells, the strip and the trees; this function decides what goes into them and adds the colour key beside them.

Usage

```
draw_heatmap(
  data,
  group = NULL,
  group_lv = NULL,
  feats = NULL,
  scale = c("feature", "sample", "none"),
  zlim = NULL,
  dist_method = c("euclidean", "correlation", "manhattan"),
  hclust_method = c("average", "complete", "ward.D2"),
  cluster_feats = TRUE,
  cluster_samples = TRUE,
  feat_labels = NULL,
  sample_labels = NULL,
  show_feat_names = TRUE,
  show_sample_names = TRUE,
  anno = FALSE,
  cex.anno = 1,
  n_colors = 101,
  main = NULL,
  cex.axis = 0.9,
  cex.main = 1.5,
  cex.legend = 1.2
)
```

Arguments

data	A data.frame or a matrix in wide format, one row per observation and one column per feature. This is the same layout compare_two_groups() and compare_multiple_groups() take. Either type is accepted and converted to the numeric matrix the drawing needs; a matrix with no column names has them made up as V1, V2 and so on, and repeated row names are kept, since a sample name is a naming choice rather than a key. There must be at least two features and two samples to draw.
group	Grouping vector with one entry per row of data, or NULL when group_lv is also NULL to draw without a group strip or group legend.
group_lv	Character vector of group levels, in the order they should appear in the legend, or NULL with group = NULL. Rows belonging to any other level are dropped when both are supplied.
feats	Character vector of numeric column names to draw, or NULL for every column of data. The order given is the order the rows are drawn in when they are not clustered.
scale	How to put the features on a comparable scale before drawing: "feature" z-scores each feature across the samples, "sample" z-scores each sample across

the features, and "none" draws the values as they arrived. The scaled values are what matrix comes back holding.

zlim — Numeric length-2 range the colours span, or NULL to derive it from the values being drawn. Values outside a supplied range are drawn at the end of the scale rather than left blank, and how many were is reported.

dist_method — Distance behind the clustering. "correlation" is 1 - cor(), which groups features by the shape of their profile rather than by how high it sits; the other two are handed to stats::dist().

hclust_method — Linkage handed to stats::hclust().

cluster_feats, cluster_samples — Whether to cluster and reorder that axis. FALSE keeps the input order.

feat_labels, sample_labels — Labels to draw in place of the column names of data and its row names. sample_labels has one entry per row of data and is filtered along with it; feat_labels has one entry per feature in feats.

show_feat_names, show_sample_names — Whether to draw those labels. The labels still exist when they are not drawn, so a hidden axis does not change anything else about the plot.

anno — If TRUE, write each cell value on the cell, rounded to two decimal places. The numbers are the values in the returned matrix (after scale, and not clamped to zlim). Missing cells are left blank.

cex.anno — Character expansion for those cell labels, relative to the smaller of the feature and sample axis label sizes.

n_colors — Number of colours in the blue-white-red ramp.

main — Plot title.

cex.axis, cex.main, cex.legend — Character expansion for the axis labels, the title and the group legend.

Details

Features are z-scored across the samples by default. Without it a single high-abundance feature takes the whole colour range and everything else is left white, since the colour scale is shared by every cell in the plot and features are not measured on a common scale. The plot does not say which scale ran: naming it would take a line of text wider than the key it titles, and the numbers beside the key are the scaled values either way. A feature with no variance would divide by zero, so it is only centred and ends up flat at the middle of the scale; how many were is reported in a message.

A diverging palette needs a meaningful midpoint. When zlim is not given, zero is that midpoint if the values being drawn have both signs, which is always the case after z-scoring, and the range is made symmetric around it. Values of one sign only have no such point, so the range is the range of the values and white falls in the middle of it.

Missing values are drawn as grey cells rather than dropped, so a gap is visible as a gap. Clustering leaves them to stats::dist(), which measures a pair on the values they share. A pair that shares none has no distance at all, and rather than fail, that axis is left in its input order with a message saying so.

The clustering is done here rather than left to stats::heatmap(), which would standardise after clustering and divide a flat feature by zero, and the trees are handed over already built. What comes back is therefore the tree the plot shows. The panel proportions are stats::heatmap()'s, including the square cells its respect = TRUE layout asks for. That layout keeps the block of panels square and centres it, so a device much wider than it is tall is left with empty space to the right of the plot and its key.

stats::heatmap() draws no colour key, so this function keeps a strip beside the plot for one and overlays it after: an upright bar with its numbers to the right of it, and the group legend to the right of those, both starting level with the top row of cells. The strip is as wide as those need rather than a fixed fraction of the device, and it is placed against the feature labels rather than at the edge, since the space the centred layout leaves over would otherwise read as a gap between the plot and its key. The panel layout, the margins and the plot region are all restored on exit, so the caller's device is left as it was found.

Value

A list, invisibly:

matrix The scaled matrix as it was drawn: features in rows, samples in columns, both in the order they appear on the plot, and the row and column names as they were labelled. Values are not clamped to zlim, which is a decision about colour rather than about the data.

feat_order,sample_order The permutations the clustering chose, as indices into feats and into the rows of data that were kept.

feat_hclust,sample_hclust The stats::hclust() objects behind the dendrograms, or NULL for an axis that was not clustered.

zlim The colour range, derived or as supplied.

group_colors The colour drawn for each group level, or NULL when no group was supplied.

See Also

draw_grouped_boxplot(), which takes the same wide input.

Examples

```
## Features are z-scored, so the two planted directions separate by colour
## and the clustering of the columns finds the two groups on its own.
sim <- simulate_two_groups(n_feats = 30, n_up = 5, n_down = 5, seed = 3)
out <- draw_heatmap(
  data     = as.matrix(sim$args$data),
  group    = sim$args$group,
  group_lv = sim$args$group_lv,
  show_sample_names = FALSE,
  main     = "30 features, 10 of them planted"
)

## The clustering is on the result, so what the picture shows can be checked
## rather than eyeballed.
head(rownames(out$matrix))
out$sample_hclust$method
```

```
## Only the planted features, in the order they were named
draw_heatmap(
  data          = sim$args$data,
  group         = sim$args$group,
  group_lv      = sim$args$group_lv,
  feats         = sim$truth$features[sim$truth$direction != "none"],
  cluster_feats = FALSE,
  show_sample_names = FALSE
)

## Unscaled values, with the four measurements on their own scale
draw_heatmap(
  data     = iris[1:4],
  group    = iris$Species,
  group_lv = levels(iris$Species),
  scale    = "none",
  show_sample_names = FALSE,
  main     = "iris, cm"
)

## No group strip or group legend when both are omitted
draw_heatmap(
  data = as.matrix(sim$args$data),
  show_sample_names = FALSE
)
```

`draw_interaction_plot` *Draw an interaction plot of a factorial comparison*

Description

Joins the cell means of one factor across the levels of another, one line per level of the tracing factor, which is the picture an interaction term is a test of: parallel lines are additive factors and crossing or fanning lines are the interaction the term reports a p-value for.

Usage

```
draw_interaction_plot(
  comparison_result,
  x = NULL,
  trace = NULL,
  facet = NULL,
  type = c("auto", "pairwise", "matrix", "facet"),
  feats = NULL,
  errorbar = c("none", "se", "ci"),
  panel_nrow = NULL,
  dark = FALSE,
```

```
  ylim = NULL,
  xlab = NULL,
  ylab = NULL,
  main = NULL,
  col = NULL,
  lwd = 2,
  cex.axis = 1.2,
  cex.lab = 1.3,
  cex.main = 1.3,
  cex.legend = 1.1
)
```

Arguments

comparison_result	A factorial comparison result, as returned by compare_factorial_groups().
x	Name of the factor on the x axis.
trace	Name of the factor one line is drawn per. Unnamed, the factors are taken in declaration order, trace first, which is the order draw_grouped_boxplot() colours a crossed design in.
facet	Name of the factor whose levels are kept in panels of their own instead of averaged away. Required by type = "facet" and refused by the other two views.
type	"auto", "pairwise", "matrix" or "facet".
feats	Character vector of features to draw, in panel order. NULL, the default, draws every feature in the pairwise view and the first feature in the other two, which is reported in a message().
errorbar	"none", "se" for one pooled standard error either side of each mean, or "ci" for a confidence interval at the conf_level the comparison was run at.
panel_nrow	Number of rows to arrange the panels in, or NULL for a default that depends on the view. Not used by the matrix view, whose grid the factors fix.
dark	If TRUE, use a dark background with light text.
ylim	Numeric length-2 y axis range, or NULL to derive it from the means and their bars.
xlab, ylab, main	Axis and title labels. All three are derived from the result when left NULL. A single xlab covers every panel of the matrix view, whose panels are each on a different factor, so leaving it NULL there is what lets each panel name its own.
col	Colours for the levels of the tracing factor, recycled if short. NULL takes them from grDevices::hcl.colors(), the palette draw_grouped_boxplot() draws a crossed design in.
lwd	Line width of the traces.
cex.axis, cex.lab, cex.main, cex.legend	Character expansion for the axis annotation, the axis label, the title and the legend.

Details

The means come from `comparison_result$cells`, so the plot and the F tests beside it describe the same fit rather than two passes over the data. Where the design holds more factors than the two being drawn, the remaining ones are averaged away, unweighted, which is the same marginal mean the post-hoc stage contrasts: the gap between two points of one line is the `estimate` its contrast reports in `comparison_result$posthoc`.

Three views are available:

`"pairwise"` One pair of factors, `trace` against `x`, with every other factor averaged away. One panel per feature, so several features can be read side by side.

`"matrix"` Every pair of factors at once, in an upper triangle: the row is the tracing factor and the column the one on the x axis. One feature only, since the panels are spent on the factors.

`"facet"` `trace` against `x` again, but with the levels of `facet` kept apart in panels of their own rather than averaged away, which is what shows a two-factor interaction that itself depends on a third factor. One feature only, for the same reason.

`type = "auto"`, the default, reads the arguments: naming `facet` asks for the facet view, naming `x` or `trace` for the pairwise one, and naming none of them gives the pairwise view for two factors and the matrix for three or more.

The function changes graphical parameters and the panel layout, and restores both on exit, so the caller's device is left as it was found.

Each level of the tracing factor gets a colour, a plotting symbol and a line type of its own, so the lines stay apart in greyscale and for a reader who cannot tell the two colours apart.

Value

The plotted means, invisibly, one row per point in the order they were drawn, with columns `panel`, `features`, `x_factor`, `x_level`, `trace_factor`, `trace_level`, `n_cells`, `mean`, `se`, `lower_conf` and `upper_conf`. `n_cells` is how many cells were averaged into the point, so `1` marks a cell mean and anything more a marginal one. The view that `type = "auto"` resolved to is attached as the attribute `"view"`.

What an error bar here is not

The bars are built from `cells$se`, which is `sqrt(ms_error / n)` pooled over the whole model, so a bar on a marginal mean over a set of cells has half-width `sqrt(sum(se^2) / length(se)^2)` times one or the `qt()` multiplier. That is the standard error of the mean being drawn, not of the difference between two of them, and the difference is what the interaction is about. Two bars that overlap do not settle the term test, which is in `$terms`, and two points whose gap you want tested are in `$posthoc`.

See Also

`compare_factorial_groups()` for the result this reads, `draw_grouped_boxplot()` for the observations behind the means, and `draw_volcano_plot()` with `estimate_significance(by = "term")` for which features have an interaction at all.

Examples

```
sim <- simulate_factorial_groups(n_feats = 4, n_per_cell = 8, seed = 1)
res <- do.call(compare_factorial_groups, c(sim$args, list(diagnose = FALSE)))

## One panel per feature, the first factor tracing the second
draw_interaction_plot(res)

## The same two factors the other way round, with standard error bars
draw_interaction_plot(res, x = "treatment", trace = "sex", errorbar = "se")

## Two features only, stacked
draw_interaction_plot(res, feats = res$features[1:2], panel_nrow = 2)

## Every pair of factors of a three-factor design, for one feature
sim3 <- simulate_factorial_groups(
  n_feats = 2, n_per_cell = 8, seed = 2,
  factor_lv = list(treatment = c("control", "treat_A"),
                   sex       = c("male", "female"),
                   site      = c("north", "south", "east"))
)
res3 <- do.call(compare_factorial_groups,
                c(sim3$args, list(diagnose = FALSE)))
draw_interaction_plot(res3, type = "matrix")

## The third factor kept apart in panels instead of averaged away
draw_interaction_plot(res3, x = "site", trace = "treatment", facet = "sex",
                      errorbar = "ci")
```

---

draw_mosaic_plot *Draw a mosaic plot of a contingency table*

Description

Splits the x axis by the first variable's marginal shares and each strip by the second variable's conditional shares, so the area of a tile is the cell's share of the table. Three things are drawn on top of that geometry, and each of them answers a question the geometry alone leaves open.

Usage

```
draw_mosaic_plot(
  categorical_comparison_result,
  shade = TRUE,
  residual = c("pearson", "standardized"),
  expected_line = TRUE,
  anno_cells = c("auto", "count", "percent", "both", "none"),
  gap = 0.015,
  xlab = NULL,
  ylab = NULL,
```

```
  main = NULL,
  cex.lab = 1.3,
  cex.axis = 1.2,
  cex.main = 1.3,
  cex.legend = 1.1,
  cex.anno = 1,
  dark = FALSE
)

## S3 method for class 'sa_categorical'
plot(x, ...)
```

Arguments

categorical_comparison_result	A categorical comparison result, as returned by compare_categorical_groups(). The levels that take part, their order and the null hypothesis the shading is read under are all settled there, so there is no category_lv or control_label to restate here.
shade	Logical. If FALSE, every tile is drawn in the background colour and the residual key is omitted.
residual	Which residual the shading reads. "pearson" squares and sums to the test statistic, so it says how the statistic was made up. "standardized" is referred to a standard normal, so it says which cells are individually surprising, and it is only defined when the null is a statement about the margins.
expected_line	Logical. If TRUE, mark each strip at the tile boundaries the null hypothesis expects.
anno_cells	What to write on a tile. "count" is the observed count, "percent" the tile's share of its own strip, "both" puts one over the other, and "none" writes nothing. "auto", the default, writes as much as the tile has room for, measured against the label rather than against a fixed fraction of the plot. TRUE and FALSE are accepted as "auto" and "none".
gap	Gap between neighbouring tiles, as a fraction of the axis, for each gap rather than for all of them together. Capped so that the gaps never take more than two fifths of an axis however many levels there are.
xlab, ylab, main	Axis labels and title. NULL takes the variable names from the table, or no title.
cex.lab, cex.axis, cex.main, cex.legend, cex.anno	Character expansion for the axis labels, the level names, the title, the residual key and the tile annotation.
dark	Logical. If TRUE, draw on a dark background with light annotation, the same palette draw_grouped_boxplot() uses.
x	A categorical comparison result, as returned by compare_categorical_groups().
...	Arguments passed on to draw_mosaic_plot().

Details

The **shading** says which cells made the statistic what it is. It reads the Pearson residual, the quantity that squares and sums to that statistic, at the conventional cuts of 2 and 4. The two colours are the ones `draw_volcano_plot()` already uses for a feature that moved up and one that moved down, so "more than expected" and "less than expected" are the same pair of colours the rest of the package reads as a direction.

The **expected line** says what "expected" was. A dashed segment sits at each boundary the tiles of a strip would have had under the null hypothesis, so the departure is the distance between a tile edge and the line beside it rather than something to be inferred by comparing strips by eye. Under independence the lines fall at the same heights in every strip, which is what makes an association visible at a glance; under symmetry they do not, because there the expectation is a cell against its own transpose.

The **annotation** says how many observations a tile stands for, which area cannot: a wide short tile and a narrow tall one can hold the same count.

The level names on the y axis are read off the **reference strip**, the first one, which the `control_label` and `category_lv` of `compare_categorical_groups()` decide. No single set of positions can label every strip, since the whole content of a mosaic is that the strips are cut at different heights, so labelling one of them and saying which is the honest version of the choice. The first strip is the one the rest of the package already treats as the reference.

Only the `par()` values this function sets are put back. A blanket `par(no.readonly = TRUE)` snapshot also carries `fin`, `pin` and `mai`, which are absolute sizes, and restoring them pins the next plot to the size this one happened to be drawn at.

Value

Invisibly, a list of the picture as it was drawn.

`cells` The cell table with `x1`, `x2`, `y1`, `y2` and `fill` added, in the order the tiles were painted.

`widths` Marginal share of each strip, named by row level. These are the widths before the gaps are taken out of them.

`heights` Conditional share of each tile within its strip, as a matrix of row level by column level.

`expected_prop` The same shares the null hypothesis expects, which is what the dashed segments were drawn from.

`empty_levels` `row` and `col` levels that hold no observation, so drew no tile and took no axis label.

`null` Which hypothesis the shading and the segments are about.

`residual`,`residual_breaks`,`colors` The scale the shading read.

Which null hypothesis is drawn

The one the result was tested against, `categorical_comparison_result$design$null`. That is the whole point of reading it off the result rather than recomputing it here: a matched design is tested for symmetry, so a mosaic of it shaded by departure from independence would be a picture of a hypothesis nothing in the result has a p-value for. A bare `table()` carries no such hypothesis, which is why one is not accepted here: `compare_categorical_groups()` is what settles the null, the levels and their order, and this function draws what it settled.

`residual = "standardized"` is refused under symmetry. The variance correction that residual divides by is derived for a table held against its own margins and `$cells$std_residual` is `NA` there, so the request has no answer rather than a different one.

See Also

`compare_categorical_groups()` for the analysis this draws, and `draw_grouped_boxplot()` for the numeric counterpart.

Examples

```
smoking <- data.frame(
  smoker = rep(c("y", "n"), each = 60),
  grade  = c(rep(c("high", "mid", "low"), c(10, 20, 30)),
             rep(c("high", "mid", "low"), c(30, 20, 10)))
)
res <- compare_categorical_groups(smoking)
draw_mosaic_plot(res)

## The dashed segments are what independence expected, so the departure is a
## distance rather than a comparison between strips.
drawn <- draw_mosaic_plot(res, anno_cells = "both")
drawn$expected_prop

## The levels that take part and the order they sit in are settled by the
## comparison, so a different reference is a different call to it.
draw_mosaic_plot(
  compare_categorical_groups(smoking, control_label = c(smoker = "y"))
)

## A matched design is shaded by departure from symmetry, so the diagonal is
## neutral by construction and only the discordant cells carry colour.
before_after <- data.frame(
  before = rep(c("pass", "fail"), c(20, 30)),
  after  = c(rep(c("pass", "fail"), c(18, 2)), rep(c("pass", "fail"), c(14, 16)))
)
draw_mosaic_plot(compare_categorical_groups(before_after, paired = TRUE))
```

---

draw_prediction_plot *Draw predicted against observed for an evaluated regression*

Description

The picture of an `evaluate_regression_models()` result: each model's predictions against the outcome they were predicting, with the identity line to read them against and the calibration line to read the identity against.

Usage

```
draw_prediction_plot(
  performance_result,
  models = NULL,
  type = c("auto", "overlay", "panel"),
  panel_nrow = NULL,
  points = TRUE,
  anno_corr = FALSE,
  anno_rsq = FALSE,
  anno_lm = FALSE,
  dark = FALSE,
  lim = NULL,
  col = NULL,
  lwd = 2,
  xlab = NULL,
  ylab = NULL,
  main = NULL,
  cex.axis = 1.2,
  cex.lab = 1.3,
  cex.main = 1.3,
  cex.legend = 1.1,
  cex.anno = NULL
)
```

Arguments

`performance_result`	A regression evaluation, as returned by `evaluate_regression_models()`.
`models`	Which models to draw and in what order, or `NULL` for all of them in the order the evaluation holds, which puts the baseline first.
`type`	`"auto"`, `"overlay"` or `"panel"`, as described above.
`panel_nrow`	Rows of panels under `type = "panel"`. `NULL` draws them in one row.
`points`	Whether to draw the predictions themselves. `FALSE` leaves the calibration lines, which is what makes a crowded overlay readable.
`anno_corr`	Whether to report each model's correlation beside its name.
`anno_rsq`	Whether to report each model's held-out R-squared from `$metrics$r_squared`. This is `1 - SSE/SST` on the scored rows, not `cor` squared, and the two are annotated separately for the same reason the table carries both.
`anno_lm`	Whether to report each model's calibration line as an equation.
`dark`	Whether to draw on a dark background.
`lim`	Range of both axes, or `NULL` to span every value drawn.
`col`	One colour, or one per drawn model. `NULL` takes them from `hcl.colors(n, "Dark 2")`.
`lwd`	Width of the calibration lines.
`xlab, ylab, main`	Axis and figure labels. `NULL` builds them from the evaluation.

cex.axis, cex.lab, cex.main, cex.legend, cex.anno
: Relative text sizes. cex.legend sizes the model-name legend; cex.anno sizes the correlation, R-squared and calibration labels drawn inside the plot area. NULL matches cex.legend.

Details

Two lines are drawn in every panel. The dotted grey one is the identity, where a prediction that was exactly right would lie. The solid coloured one is lm(predicted ~ observed), taken from the calib_slope and calib_intercept of $metrics rather than fitted again here, so the picture and the table cannot drift apart. A slope under one is a model whose predictions are squeezed towards their own mean, which is the usual shape of a fit scored on rows it has not seen.

type decides how more than one model is shown. "panel" gives each its own square, which is the default past one model because two clouds of points on shared axes make a third cloud that belongs to neither. "overlay" puts them all in one square, which is readable with points = FALSE, where what is compared is the calibration lines. "auto" is "overlay" for a single model and "panel" beyond that.

Both axes span the same range in every panel, taken over every model drawn, so the panels are comparable and the identity line is the diagonal of the square rather than an arbitrary chord.

Value

The rows of $metrics that were drawn, invisibly, carrying the resolved type as a "view" attribute.

See Also

evaluate_regression_models() for the result this draws, and draw_roc_curve() for the classification counterpart.

Examples

```
train <- mtcars[1:24, ]
test <- mtcars[25:32, ]
full <- fit_linear_regression(train, outcome = "mpg",
                              predictors = c("wt", "hp", "disp"),
                              cv = FALSE)
small <- fit_linear_regression(train, outcome = "mpg",
                               predictors = "wt", cv = FALSE)
res <- evaluate_regression_models(full, list(weight_only = small),
                                  newdata = test)

draw_prediction_plot(res, anno_lm = TRUE)
draw_prediction_plot(res, type = "overlay", points = FALSE,
                     anno_corr = TRUE, anno_rsq = TRUE)
```

`draw_roc_curve` *Draw the ROC curves of an evaluated classification*

Description

The picture of an `evaluate_classification_models()` result: one curve per model, all on the rows every model was scored on, with the chance diagonal to read them against.

Usage

```
draw_roc_curve(
  performance_result,
  models = NULL,
  anno_auc = FALSE,
  chance = TRUE,
  dark = FALSE,
  col = NULL,
  lwd = 2,
  lty = 1,
  legend_pos = "bottomright",
  xlab = NULL,
  ylab = NULL,
  main = NULL,
  cex.axis = 1.2,
  cex.lab = 1.3,
  cex.main = 1.3,
  cex.legend = 1.1,
  cex.anno = NULL
)
```

Arguments

`performance_result`	A classification evaluation, as returned by `evaluate_classification_models()`.
`models`	Which models to draw and in what order, or `NULL` for all of them in the order the evaluation holds, which puts the baseline first.
`anno_auc`	Whether to add each model's AUC to its legend entry.
`chance`	Whether to draw the chance diagonal.
`dark`	Whether to draw on a dark background.
`col`	One colour, or one per drawn model. `NULL` takes them from `hcl.colors(n, "Dark 2")`.
`lwd`	Width of the curves.
`lty`	One line type, or one per drawn model.
`legend_pos`	Where to put the legend, or `NULL` for no legend.

`xlab`, `ylab`, `main` Axis and figure labels. `NULL` builds them from the evaluation.

`cex.axis`, `cex.lab`, `cex.main`, `cex.legend`, `cex.anno`
Relative text sizes. `cex.legend` sizes the model-name legend when `anno_auc = FALSE`. `cex.anno` sizes it once each entry carries an AUC; `NULL` matches `cex.legend`.

Details

The curves are `$curves`, the operating points the evaluation already computed, rather than anything recomputed here, so the picture and the `auc` column of `$metrics` describe the same curve. Consecutive points are joined by straight lines, which is what makes the area under the drawn curve the `auc` beside it: a run of tied predictions cannot be separated by any threshold and the curve crosses it diagonally.

Every model is drawn against the same rows, since that is what the evaluation scored them on, so two curves crossing is a statement about the models rather than about which rows each of them managed.

Value

The rows of `$metrics` that were drawn, invisibly.

See Also

`evaluate_classification_models()` for the result this draws, and `draw_prediction_plot()` for the regression counterpart.

Examples

```
iris2 <- iris[iris$Species != "setosa", ]
iris2$Species <- factor(iris2$Species)
train <- iris2[c(1:35, 51:85), ]
test <- iris2[c(36:50, 86:100), ]

full <- fit_logistic_regression(train, outcome = "Species", cv = FALSE)
petal <- fit_logistic_regression(train, outcome = "Species",
                                 predictors = "Petal.Width", cv = FALSE)
res <- evaluate_classification_models(full, list(petal_only = petal),
                                      newdata = test)

draw_roc_curve(res, anno_auc = TRUE)
```

`draw_volcano_plot` *Draw a volcano plot*

Description

Plots log2fc against -log10(pvalue) for the table estimate_significance() returns, colours the features that clear both cutoffs by direction and labels the strongest of them. A term reading of a factorial comparison is drawn as one panel per term in a single figure.

Usage

```
draw_volcano_plot(
  significance_result,
  terms = NULL,
  panel_nrow = NULL,
  use_adjusted = TRUE,
  log2fc_cutoff = NULL,
  pval_cutoff = NULL,
  anno_feats = TRUE,
  anno_top = 10,
  cex.anno = 1,
  xlim = NULL,
  ylim = NULL,
  xlab = NULL,
  main = NULL,
  cex.lab = 1.3,
  cex.axis = 1.2,
  cex.main = 1.3,
  margin = c(5, 5, 4, 3),
  ...
)
```

Arguments

significance_result	The object returned by estimate_significance(), whose significance element is what is plotted. With by = "contrast" that element holds one table per contrast, so name the one to draw: sig$significance[["virginica - setosa"]]. With by = "term" the whole list is drawn, one panel per term. A bare verdict data.frame is accepted too.
terms	Which terms get a panel, read only under a term reading. NULL, the default, draws the two main effects of the first two factors and their interaction, which is every term of a two-factor design and three of the seven of a three-factor one; the terms left out are named in a message(). Give term labels, as $terms$terms spells them, to draw those instead.
panel_nrow	How many rows the panels are laid out in, or NULL for a single row, which is what three panels of a two-factor design want.
use_adjusted	Logical. If TRUE, plot and threshold the adj_pvalue column; if FALSE, use the unadjusted pvalue column. The axis label follows, so the y axis always describes what was actually plotted.

log2fc_cutoff, pval_cutoff
: Cutoffs for calling a feature changed and significant, drawn as guides. NULL, the default, takes the values estimate_significance() recorded on significance_result, so the guides agree with its is_signif column. Supply a number to override them. Selecting a subset of rows keeps the recorded values, but selecting columns drops them, in which case both must be given.

anno_feats
: Logical. If TRUE, label the strongest significant features. A run where no feature clears both cutoffs still draws the plot, with a message() in place of the labels.

anno_top
: How many features to label in each direction, so up to 2 * anno_top labels in total.

cex.anno
: Character expansion for those labels.

xlim, ylim
: Numeric length-2 axis ranges, or NULL to derive them from the data. A supplied range is used as given. Across panels a derived range is derived from all of them at once, so the panels can be compared.

xlab
: X axis label, or NULL to derive it from what log2fc compares in the comparison behind the verdict.

main
: Plot title. With panels it is the title of the figure, written once above them, and each panel is titled with its term.

cex.lab, cex.axis, cex.main
: Character expansion for the axis labels, the axis annotation and the title.

margin
: Plot margins in lines, passed to graphics::par() as mar. Applied to every panel.

...
: Additional arguments passed to graphics::plot().

Details

The plot margins are restored on exit, but the coordinate system is deliberately left in place, so graphics::points(), graphics::text() and friends can still be used to add to the finished plot. With panels what is left in place is the last panel's coordinate system, and the panel grid itself is undone, mfrow included, since a figure of several plots is finished when the last of them is drawn.

A p-value of exactly zero has no finite -log10(). Rather than dropping the most significant points or letting an infinite axis limit blank the plot, the axis is scaled to the largest finite value and those points are drawn at the top of it. Non-finite log2fc, which compare_two_groups() produces when a group centre is zero, is capped at the edge of the x axis the same way. Either kind of capping is reported in a message(), since a capped point no longer sits at its true coordinate.

What log2fc compares is not the same question in every scenario, so the x axis label follows the comparison the verdict came from. A multi-group omnibus verdict carries the single fold change its effect table holds, the level furthest from the reference rather than any named pair, and the label says so. A factorial omnibus verdict does the same with cells: the reference cell is where every factor sits at its first level, and the label names it. A term table carries an ANOVA component rather than a ratio of two centres, so it is labelled as an effect and not as a fold change; see "The size of a term" in compare_factorial_groups() for how to read its magnitude. Every other reading compares two fixed centres and is labelled log2 FC. Pass xlab to override this.

Points are coloured by the same masks that select the labels, so which features are highlighted and which are labelled can never disagree. With the default arguments those masks reproduce the is_signif column of the input. The labels are drawn in a brighter shade than the points on purpose, so that a label stays legible where it overlaps them.

Value

NULL invisibly.

One panel per term

A term reading is drawn as a figure rather than asked to name one table, unlike a contrast reading. A crossed design decomposes into a fixed and small set of terms, three for two factors, and the whole point of the reading is to see which of them a feature responded to, which is a comparison between the panels. The number of pairwise contrasts, by contrast, follows from the level counts and is arbitrary.

Every panel is judged by one rule, the cutoffs of the first table, and shares the axes with the others unless xlim or ylim says otherwise. Both are what makes a point in one panel comparable with a point in another.

See Also

estimate_significance(), whose output is the only argument this function needs.

Examples

```
iris2 <- iris[iris$Species != "setosa", ]
res <- compare_two_groups(
  data     = iris2,
  feats    = c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width"),
  group    = iris2$Species,
  group_lv = c("virginica", "versicolor")
)
sig <- estimate_significance(res, log2fc_cutoff = 0.1)

draw_volcano_plot(sig, main = "virginica vs versicolor")

## Unadjusted p-values, no labels
draw_volcano_plot(sig, use_adjusted = FALSE, anno_feats = FALSE)

## A cutoff other than the one the verdict used
draw_volcano_plot(sig, log2fc_cutoff = 0.3)

## A multi-group verdict says on the x axis which levels its log2fc compares
multi <- compare_multiple_groups(iris, c("Sepal.Length", "Petal.Length"),
                                 iris$Species, levels(iris$Species))
draw_volcano_plot(estimate_significance(multi, log2fc_cutoff = 0.1))

## A factorial omnibus verdict names the reference cell on the x axis
fact <- compare_factorial_groups(
  data    = warpbreaks,
  feats   = "breaks",
  factors = list(wool = "wool", tension = "tension"),
  posthoc = FALSE
)
draw_volcano_plot(estimate_significance(fact, log2fc_cutoff = 0.1),
                  main = "warpbreaks")
```

```
## One panel per term of a crossed design: wool, tension, wool:tension
draw_volcano_plot(estimate_significance(fact, by = "term", log2fc_cutoff = 0.1),
                  main = "warpbreaks, by term")
```

`estimate_categorical_significance` *Reduce a contingency table to significance verdicts*

Description

The categorical counterpart of `estimate_significance()`, and the reason it is a second function rather than a branch of that one is that the two axes a verdict is made of sit at a different granularity here. A comparison asks its question once per feature; a contingency table is asked about as a whole, and the place where a signed effect and a p-value exist side by side is one level down, at the cell.

Usage

```
estimate_categorical_significance(
  categorical_comparison_result,
  by = c("cell", "table"),
  test = names(categorical_comparison_result$tests)[1],
  log2_lift_cutoff = 1,
  pval_cutoff = 0.05,
  adj_type = "BH",
  measure = "auto",
  effect_cutoff = NULL
)
```

Arguments

`categorical_comparison_result`
: A categorical comparison result, as returned by `compare_categorical_groups()`. A numeric comparison is refused and pointed at `estimate_significance()`.

`by`
: `"cell"` for one verdict per cell of the table, `"table"` for one verdict for the table as a whole. See "The two readings".

`test`
: Which test supplies the p-value of a table reading. One of `names(categorical_comparison_result$tes` so `"chisq_test"` or `"fisher_test"` for an independent design. Defaults to the first test the design ran. Read only by `by = "table"`: a cell reading takes its p-value from the cell's own standardized residual, which no test reports.

`log2_lift_cutoff`
: Minimum `abs(log2_lift)` required to call a cell significant. Read only by `by = "cell"`. The default of 1 is a cell holding twice, or half, what its null expected, and is deliberately the same number `estimate_significance()` uses. It is a strict demand of a contingency table, where departures well under a doubling are ordinary; halve it to ask of a cell what the default asks of a fold change.

`pval_cutoff` Largest p-value allowed for a verdict of significant. Read by both readings, against `adj_pvalue` under `by = "cell"` and against the test's own `pvalue` under `by = "table"`.

`adj_type` Multiplicity adjustment across the cells, one of stats::p.adjust.methods. Read only by `by = "cell"`. Unlike the `adj_type` of `estimate_significance()` this is the first adjustment rather than a replacement for one, since a categorical result carries no adjusted column. `"none"` tests the raw p-values.

`measure` Which row of `$association` a table reading puts on its effect axis, or `"auto"` to take the one the design defines. Read only by `by = "table"`.

`effect_cutoff` Magnitude `measure` has to reach for a table reading to call the table significant, or `NULL` to judge on the p-value alone. Read only by `by = "table"`.

Details

`is_signif` is three-valued and follows the rule the numeric scenarios use. `NA` is undecided rather than decided against, which is what an `NA` `std_residual` or an undefined `lift` leaves a cell as.

A cell holding no observation at all has a `lift` of exactly zero and so a `log2_lift` of `-Inf`, an infinitely large shortfall, which clears any magnitude cutoff. A cell whose expected count is zero has no ratio to take and is `NA` in both columns. The two are different findings and are reported differently, the same way `estimate_significance()` separates a fold change of zero from one that cannot be formed.

Value

An `sa_categorical_significance` object, a list of two elements:

`analysis_type` `"categorical_comparison"`.

`significance` With `by = "cell"`, a data.frame with one row per cell and the columns `row_level`, `col_level`, `observed`, `expected`, `lift`, `log2_lift`, `std_residual`, `pvalue`, `adj_pvalue` and `is_signif`. With `by = "table"`, a one-row data.frame with `measure`, `estimate`, `lower_conf`, `upper_conf`, `pvalue` and `is_signif`.

The cutoffs, the reading, the null hypothesis and whichever of the test name and the adjustment applies are attached to the data.frame as attributes, so a table that has been passed around still says which rule produced it.

A cell table keys on `c("row_level", "col_level")`, the key `categorical_comparison_result$cells` and `simulate_categorical_groups()$truth_cell` also use, so a verdict merges with either without renaming. Note that `truth_cell` carries a `lift` column of its own, the planted one, so a merge of the two distinguishes them as `lift.x` and `lift.y`.

This is deliberately not an `sa_significance`. Its columns are cells rather than features and `log2_lift` rather than `log2fc`, so `draw_volcano_plot()` refusing it is the point of the separate class rather than an omission. `draw_mosaic_plot()` is what draws this scenario.

Why a cell has both axes and a table has one

`compare_categorical_groups()` says that an association is not signed, which is true of the **table**: `cramers_v` reports how far the table sits from its null and not in which direction, because past a

2 x 2 there is no single direction to name. A **cell** is different. It was expected at some count and observed at another, and `observed / expected` says both how far it moved and which way.

That ratio is `lift`, the same quantity `simulate_categorical_groups()` plants and reports in `truth_cell$lift`, and `log2(lift)` is what the effect axis of this function is. It is defined on every table, 2 x 2, 2 x 3 or larger, so one cutoff means the same thing whatever shape the table is.

The p-value axis is `std_residual`, which is built to be referred to a standard normal, so `2 * pnorm(-abs(std_residual))` is the cell's own two-sided p-value. This is the standard post-hoc reading of a contingency table.

The two axes are not the same information. `lift` is a ratio and does not change when the table is observed on twice as many rows; `std_residual` grows with the square root of the count. So a cell can be far from what was expected and poorly evidenced, or close to it and firmly established, exactly as `log2fc` and a p-value come apart in the numeric scenarios.

A cell axis also restores a multiplicity axis. The table has as many cells as it has, they are one family, and `adj_type` adjusts across them. That is why this function computes an adjustment rather than reusing one: `sa_categorical` carries no `pval_adj` column, there being nothing to adjust across at the level the tests are reported at.

The two readings

`by = "cell"` One row per cell of the table, with `log2_lift` on the effect axis and the adjusted p-value of the cell's standardized residual on the other. This is the reading that generalises, so it is the default.

`by = "table"` One row, the whole-table verdict: an association measure out of `$association` beside the p-value of one of the tests. There is no multiplicity here, one table being one question, so no `adj_pvalue` column is carried.

Each reading ignores the arguments the other one reads, and says so rather than letting a setting that changes nothing pass unremarked.

Which measure a table reading reports

`measure = "auto"` reads the design, since which measures exist at all depends on it and on the size of the table.

design	table	measure
independent	2 x 2	`odds_ratio`
independent	larger	`cramers_v`
matched, two conditions	2 x 2	`odds_ratio_paired`
matched, three or more	condition by response	`kendalls_w`

Any other row of `$association` can be named instead.

`effect_cutoff` is `NULL` by default, and then the verdict is the p-value alone. The conventional thresholds for `cramers_v` are not defaults here because they are conventions rather than facts about the measure, and a default is the one place a convention is hardest to notice. Naming a number reads it on the measure's own scale: `odds_ratio` and `odds_ratio_paired` are ratios centred at 1, so the cutoff is a fold either way and has to be at least 1, while every other measure is centred at zero and the cutoff is compared against the magnitude.

Why a matched pair of conditions has no cell reading

A matched two-condition design is tested for symmetry, and the variance correction the standardized residual divides by is derived for a table held against its own margins. `$cells$std_residual` is therefore `NA` throughout such a result, so there is no p-value axis to read and the request is refused rather than answered with a different quantity. `by = "table"` reads it, and `draw_mosaic_plot()` refuses `residual = "standardized"` there for the same reason.

Three or more matched conditions are a different case. Their null is marginal homogeneity, read off a condition-by-response table whose arithmetic is that of independence, so the standardized residual exists and the cell reading works.

References

Haberman, S. J. (1973). The analysis of residuals in cross-classified tables. *Biometrics*, 29(1), 205-220.

Beasley, T. M. and Schumacker, R. E. (1995). Multiple regression approach to analyzing contingency tables: Post hoc and planned comparison procedures. *The Journal of Experimental Education*, 64(1), 79-93.

Agresti, A. (2002). *Categorical Data Analysis*, 2nd ed. Wiley.

See Also

`compare_categorical_groups()` for the input, `draw_mosaic_plot()` to draw the same residuals this reads, and `estimate_significance()` for the numeric scenarios.

Examples

```
smoking <- data.frame(
  smoker = rep(c("y", "n"), each = 60),
  grade  = c(rep(c("high", "mid", "low"), c(10, 20, 30)),
             rep(c("high", "mid", "low"), c(30, 20, 10)))
)
res <- compare_categorical_groups(smoking)

## One verdict per cell. The default cutoff is a doubling either way, which on
## this table only the two corners reach.
sig <- estimate_categorical_significance(res)
sig
sig$significance

## `lift` is what the effect axis is built from, and it is a ratio: the cell
## holding half of what independence expected is at 0.5.
sig$significance[c("row_level", "col_level", "lift", "log2_lift")]

## A gentler magnitude cutoff asks less of a cell than a doubling.
estimate_categorical_significance(res, log2_lift_cutoff = 0.5)

## The whole-table verdict instead. A 2 x 3 table has no odds ratio, so the
## measure `"auto"` reports is Cramer's V.
estimate_categorical_significance(res, by = "table")$significance
```

```
## Two levels of `grade` make a 2 x 2 table, which is where an odds ratio
## exists, and `effect_cutoff` is then read as a fold either way.
two_by_two <- compare_categorical_groups(
  smoking,
  category_lv = list(smoker = c("n", "y"), grade = c("low", "high"))
)
estimate_categorical_significance(two_by_two, by = "table",
                                  effect_cutoff = 2)$significance

## The cell axis against the association that was planted on it.
sim <- simulate_categorical_groups(n_samples = 400, assoc = 0.4, seed = 1)
fit <- do.call(compare_categorical_groups, sim$args)
scored <- merge(estimate_categorical_significance(fit)$significance,
                sim$truth_cell, by = c("row_level", "col_level"))
scored[c("row_level", "col_level", "lift.x", "lift.y")]
```

`estimate_significance` *Reduce a comparison to one significance verdict per feature*

Description

Puts the two axes of a volcano plot side by side and flags the features that clear both cutoffs. The effect size comes from the comparison's `effect` table and the p-value from whichever test is named, so a single comparison can be read out through the parametric, the rank-based or the robust lens without recomputing anything.

Usage

```
estimate_significance(
  comparison_result,
  test = names(comparison_result$tests)[1],
  log2fc_cutoff = 1,
  pval_cutoff = 0.05,
  adj_type = NULL,
  by = c("omnibus", "contrast", "term")
)
```

Arguments

`comparison_result`
: A comparison result, as returned by `compare_two_groups()`, `compare_multiple_groups()`, `compare_one_sample()` or `compare_factorial_groups()`. A categorical comparison is refused: it has no feature axis, which is what this function reads a verdict along. `estimate_categorical_significance()` is its counterpart and reads the cell axis instead.

test	Which test in comparison_result supplies the p-values. One of names(comparison_result$tests), so "t_test", "wilcox_test" or "robust_test" for a two-group comparison. Defaults to the first test the scenario ran, which is the parametric one in every scenario.
log2fc_cutoff	Minimum abs(log2fc) required to call a feature significant. The default of 1 is a two-fold change.
pval_cutoff	Largest adj_pvalue allowed for a feature to be called significant.
adj_type	Multiplicity adjustment. NULL, the default, reuses the pval_adj column that compare_two_groups() already computed. Naming a method from stats::p.adjust.methods re-adjusts the raw p-values instead, which is the only way to get a different adjustment without rerunning the comparison. Passing a method here does not adjust twice: the input is always the unadjusted pval column.
by	Which p-value the verdict is read from. "omnibus", the default, uses the named test's own table. "contrast" uses the pairwise stage of a multi-group comparison and returns one verdict table per contrast. "term" uses the $terms table of a factorial comparison and returns one verdict table per model term, each carrying that term's own effect size in log2fc.

Details

is_signif combines abs(log2fc) >= log2fc_cutoff with adj_pvalue <= pval_cutoff, and is therefore judged on the adjusted p-values. Pass adj_type = "none" to test the raw ones.

The two ways of reading a multi-group comparison answer different questions and use different numbers. by = "omnibus" asks whether a feature differs across the levels at all, and pairs that with the one log2fc the effect table carries, the most extreme level against the reference. by = "contrast" asks the same question of one pair of levels at a time, and each table carries the log2fc of that pair, in the direction its contrast label reads. Both divide by the reference, so the two readings agree on which way a feature moved.

The adjustment axis differs too. Under by = "contrast" with adj_type = NULL the pval_adj of the pairwise stage is reused, which was adjusted across the contrasts within each feature by posthoc_p_adjust, or not at all for Tukey's HSD and Games-Howell, whose p-values are already family-wise. Naming a method instead adjusts across the features within each contrast, which is the axis by = "omnibus" always works on.

A factorial comparison has a third reading. The default "omnibus" pairs the whole-model F test with the most extreme **cell** against the reference cell, the combination where every factor sits at its first level. The table also carries extreme_cell, naming which cell was furthest on the log2 scale. That says a feature responded to the design and how far it moved at its furthest point, but not **which part** of the design it responded to. by = "term" answers that, one table per main effect and per interaction, and needs no choice of adjustment axis: both branches adjust across the features of one term, which is the family $terms$pval_adj was built over as well.

The log2fc of a term table is $terms$log2_effect, an ANOVA component rather than a ratio of two centres. It measures a deviation from what the rest of the model predicts, so a two-level factor whose levels differ by one log2 unit contributes -0.5 and +0.5 rather than 1. The default log2fc_cutoff = 1 is therefore a stricter demand here than the same number is elsewhere; halving it asks of a two-level factor what the default asks of a fold change. See "The size of a term" in compare_factorial_groups().

A feature whose omnibus test did not clear posthoc_alpha was never compared pairwise, so its pvalue is NA in every contrast table. Its log2fc is still reported, since a ratio of group centres does not depend on a test having been run, and is_signif follows the same three-valued rule it does everywhere else: NA, undecided, unless the magnitude cutoff already rules the feature out, which makes it FALSE whatever the p-value would have been.

The two ways a fold change can fall outside the domain of log2() are not equivalent. A fold change of exactly zero gives log2fc = -Inf, an infinitely large decrease, which clears any magnitude cutoff. A fold change whose two group centres have opposite signs gives log2fc = NaN, which makes is_signif NA: such a feature is undecided rather than decided against. compare_two_groups() reports both kinds when it builds the effect table. Note that subset() and [drop NA rows silently, so filter with which(x$significance$is_signif) if the count matters.

Value

An sa_significance object, a list of two elements:

analysis_type The analysis of the comparison this verdict was read from, so a table that has been passed around still says which scenario produced it.

significance With by = "omnibus", a data.frame with one row per feature and the columns features, log2fc, pvalue, adj_pvalue and is_signif. A multi-group omnibus table also carries extreme_level, and a factorial one carries extreme_cell, naming the level or cell whose centre produced log2fc. With by = "contrast", a list of those same data.frames, one per pairwise contrast and named after it, in the order comparison_result$pairwise[[test]] fixes. With by = "term", the same again, one per model term and named after it, in the order comparison_result$terms lists them.

The cutoffs, the test name and the adjustment actually used are attached to each data.frame as attributes, which is where draw_volcano_plot() picks them up so that the plotted guides cannot disagree with the verdict. A contrast table carries contrast, group1 and group2 on top of those and a term table carries term and term_order, so a single element of significance can be handed straight to draw_volcano_plot() — and a whole list of term tables can, which is how the term panels are drawn.

See Also

compare_two_groups() for the input and draw_volcano_plot() for the output.

Examples

```
iris2 <- iris[iris$Species != "setosa", ]
res <- compare_two_groups(
  data     = iris2,
  feats    = c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width"),
  group    = iris2$Species,
  group_lv = c("virginica", "versicolor")
)

estimate_significance(res)

## The same comparison judged on the rank-based and the robust test
```

```
estimate_significance(res, test = "wilcox_test", log2fc_cutoff = 0.1)
estimate_significance(res, test = "robust_test", log2fc_cutoff = 0.1)

## A stricter fold change cutoff leaves nothing significant here, since the
## two species are well under two-fold apart on every measurement.
estimate_significance(res, log2fc_cutoff = 1.5)

## Bonferroni instead of the Benjamini-Hochberg the comparison used
estimate_significance(res, adj_type = "bonferroni", log2fc_cutoff = 0.1)

## One verdict table per pairwise contrast of a multi-group comparison
multi <- compare_multiple_groups(iris, c("Sepal.Length", "Petal.Length"),
                                 iris$Species, levels(iris$Species))
by_pair <- estimate_significance(multi, by = "contrast",
                                 log2fc_cutoff = 0.1)
names(by_pair$significance)
by_pair$significance[["virginica - setosa"]]

## One verdict table per term of a crossed design
fact <- compare_factorial_groups(
  data    = warpbreaks,
  feats   = "breaks",
  factors = list(wool = "wool", tension = "tension"),
  posthoc = FALSE
)
draw_volcano_plot(estimate_significance(fact, log2fc_cutoff = 0.1))
by_term <- estimate_significance(fact, by = "term", log2fc_cutoff = 0.1)
names(by_term$significance)
by_term$significance[["wool:tension"]]
```

`evaluate_classification_models` *Score fitted classifications on held-out rows*

Description

Predicts one or more fitted two-class models on the same rows and reports how each one discriminated, with three tests of each model against a baseline where more than one was passed. Every model is read through `predict.sa_model()` with `type = "response"`, so the five fitting functions are interchangeable here and a logistic regression, a penalized one, a forest and a machine are scored by the same arithmetic.

Usage

```
evaluate_classification_models(
  baseline_model,
  new_models = NULL,
  newdata,
```

```
  answer = NULL,
  outcome_lv = NULL,
  control_label = NULL,
  threshold = 0.5,
  conf_level = 0.95,
  baseline_label = "baseline"
)
```

Arguments

baseline_model	The reference model, as returned by fit_logistic_regression(), fit_elastic_net(), fit_rf() or fit_svm() with a two-class outcome. It is the first row of every table and what $comparisons is measured against.
new_models	Named list of further models to hold against it, such as list(selected = fit_2, penalized = fit_3), or NULL to score the baseline on its own. The names are what the tables and the plot legend call the models, so an unnamed list is refused rather than numbered.
newdata	The rows to score, typically the test half of a split_data() result. Columns no model was fitted on are ignored.
answer	The observed classes, either the name of a column of newdata or a vector with one entry per row. NULL reads the column the models were fitted to, which is the usual case.
outcome_lv	The two classes, reference first, checked against the order the models were fitted with rather than used to change it. NULL takes theirs.
control_label	The reference class on its own, checked the same way.
threshold	Where to cut the predicted probability for accuracy, sensitivity and specificity. A row is called an event when its probability is greater than or equal to this.
conf_level	Confidence level of every interval in the result.
baseline_label	What to call the baseline in the tables and the legend.

Details

The rows are the intersection rather than the union. predict() on a model answers NA for a row that is incomplete across *that model's* predictors, so models fitted on different predictor sets come back with different rows filled in. Scoring each on whatever it managed would put two AUCs from two samples in one table, and all three comparisons below are **paired** statistics that have no meaning at all across different rows. A row any model cannot predict is therefore left out of all of them, with one message saying how many went and why.

The direction is the fits'. outcome_lv[2] is the class predict(type = "response") reports the probability of, so it is the class every number here is about: sensitivity is measured against it, and an AUC above 0.5 means the models rank it above the reference. outcome_lv and control_label are read as a statement to be checked rather than as an instruction, since a fitted model cannot be re-pointed after the fact, and naming the other class is an error rather than a silent reversal. All the models must agree on it too, which evaluate_regression_models() has no counterpart of.
$metrics reports what needs no threshold first. auc comes with the interval its own DeLong standard error gives, which is a Wald interval on a bounded quantity and can therefore run past 1 for

a strong classifier; brier is the mean squared distance between the probability and the outcome, which an AUC is blind to, since a model that ranks perfectly and predicts every event at 0.6 has an AUC of 1. accuracy, sensitivity and specificity do need one, and it is threshold, recorded in $parameters so that the table says what it was measured at.
$comparisons asks three different questions of the same pair of models, which is why all three are reported rather than one being chosen.

delta_auc Whether the ranking improved, tested by DeLong's paired test. Blind to any change that does not reorder rows.

idi How much further apart the two classes' predicted probabilities moved, on the probability scale. Sees exactly the change an AUC does not.

nri How often a probability moved the right way, counting direction only, so a model that helps many rows slightly and hurts a few badly scores well here and can score badly on the IDI. Category-free: no risk strata are named, since their cut points are a clinical convention rather than a property of the data.

Every one of them is new - baseline, and every one is positive for a new model that did better.

Value

An object of class sa_performance, a plain list of nine elements.

analysis "classification_performance".

models Model names, the baseline first, in the row order every table follows.

design What was scored: the outcome label, its type, its outcome_lv, the baseline name, the row counts n_obs, n_used and n_dropped, and n_events, how many scored rows were outcome_lv[2].

parameters threshold and conf_level.

predictions One row per model and scored row: model, row (the position in newdata), observed (1 for outcome_lv[2], 0 for the reference) and predicted (the probability of outcome_lv[2]).

metrics One row per model: n_used, n_events, auc with its interval, brier, and accuracy, sensitivity and specificity at threshold.

comparisons One row per model other than the baseline, holding delta_auc, idi and nri, each with its interval and p-value, and the two class-wise components nri_event and nri_nonevent. Absent when nothing was compared.

curves The ROC operating points, one row per model and distinct predicted value: model, threshold, sensitivity, specificity. Each curve opens at threshold = Inf, the cut above every prediction, so that it starts at the corner where nothing is called an event.

metadata Package version, R version, platform and timestamp.

References

DeLong, E. R., DeLong, D. M. and Clarke-Pearson, D. L. (1988). Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach. *Biometrics*, 44(3), 837-845.

Pencina, M. J., D'Agostino, R. B., D'Agostino, R. B. and Vasan, R. S. (2008). Evaluating the added predictive ability of a new marker: from area under the ROC curve to reclassification and beyond. *Statistics in Medicine*, 27(2), 157-172.

See Also

evaluate_regression_models() for the continuous counterpart, draw_roc_curve() for the picture of this result, and predict.sa_model() for the call every model is read through.

Examples

```
## Two models of the same two-class outcome, scored on held-out rows.
iris2 <- iris[iris$Species != "setosa", ]
iris2$Species <- factor(iris2$Species)
train <- iris2[c(1:35, 51:85), ]
test <- iris2[c(36:50, 86:100), ]

full <- fit_logistic_regression(train, outcome = "Species", cv = FALSE)
petal <- fit_logistic_regression(train, outcome = "Species",
                                 predictors = "Petal.Width", cv = FALSE)

res <- evaluate_classification_models(full, list(petal_only = petal),
                                      newdata = test)
res
res$metrics
```

---

evaluate_regression_models

Score fitted regressions on held-out rows

Description

Predicts one or more fitted regressions on the same rows and reports how each one did, with the differences against a baseline where more than one model was passed. Every model is read through predict.sa_model(), so the five fitting functions are interchangeable here and a linear model, a penalized one, a forest and a machine are scored by the same arithmetic.

Usage

```
evaluate_regression_models(
  baseline_model,
  new_models = NULL,
  newdata,
  answer = NULL,
  baseline_label = "baseline"
)
```

Arguments

baseline_model	The reference model, as returned by fit_linear_regression(), fit_elastic_net(), fit_rf() or fit_svm(). It is the first row of every table and what $comparisons subtracts.

new_models	Named list of further models to hold against it, such as list(selected = fit_2, penalized = fit_3), or NULL to score the baseline on its own. The names are what the tables and the plot legend call the models, so an unnamed list is refused rather than numbered.
newdata	The rows to score, typically the test half of a split_data() result. Columns no model was fitted on are ignored.
answer	The observed outcome, either the name of a column of newdata or a vector with one entry per row. NULL reads the column the models were fitted to, which is the usual case; a model fitted from a vector remembers no name to look up and requires this.
baseline_label	What to call the baseline in the tables and the legend.

Details

The rows are the intersection rather than the union. predict() on a model answers NA for a row that is incomplete across *that model's* predictors, so a baseline fitted on nine columns and a reduced model fitted on four disagree about any row missing one of the extra five. Scoring each model on whatever it managed would put two numbers from two samples in one table and call their difference an improvement, so a row that any model cannot predict is left out of all of them, and a single message reports how many went and why. design$n_used and design$n_dropped record the outcome of that.

Every model must have been fitted to the same outcome, and to a continuous one. A classification is refused by name and pointed at evaluate_classification_models() rather than being scored by correlating a probability against a class label, which would produce a number.
$metrics reports cor and r_squared side by side because they answer different questions and agree only for predictions that need no calibration. r_squared is 1 - SSE/SST on these rows, the fraction of the variance the predictions actually removed, and it is negative for a model that does worse than the mean of the outcome. cor^2 is what that would be if the predictions were first rescaled by a line fitted to these same rows, so the gap between the two is what calib_slope and calib_intercept describe: the line is lm(predicted ~ observed), the same orientation draw_prediction_plot() draws, so a slope under one is a model whose predictions are compressed towards their own mean.

rmse and mae carry the names fit_linear_regression() and the rest use in $performance, so a resampled score and a held-out score read in the same unit under the same name. They are not the same number and are not meant to be: one is measured inside the folds of the training data and the other on rows drawn away from it.
$comparisons is every column of $metrics as new - baseline, so a positive delta_cor and a **negative** delta_rmse both say the new model did better. There is no p-value beside them, since a difference of held-out errors has no null this function is in a position to state.

Value

An object of class sa_performance, a plain list of eight elements.

analysis "regression_performance".

models Model names, the baseline first, in the row order every table follows.

`design` What was scored: the `outcome` label, its type, the `baseline` name, and the row counts `n_obs`, `n_used` and `n_dropped`.

`parameters` Empty. Scoring a regression takes no choices, which is the difference between this slot here and in a classification evaluation.

`predictions` One row per model and scored row: `model`, `row` (the position in `newdata`), `observed` and `predicted`.

`metrics` One row per model: `n_used`, `cor`, `r_squared`, `rmse`, `mae`, `bias`, `calib_slope` and `calib_intercept`.

`comparisons` One row per model other than the baseline, holding `delta_cor`, `delta_r_squared`, `delta_rmse` and `delta_mae` as `new - baseline`. Absent when nothing was compared.

`metadata` Package version, R version, platform and timestamp.

See Also

`evaluate_classification_models()` for the two-class counterpart, `draw_prediction_plot()` for the picture of this result, and `predict.sa_model()` for the call every model is read through.

Examples

```
## Two models of the same outcome, scored on rows neither was fitted on.
train <- mtcars[1:24, ]
test <- mtcars[25:32, ]

full <- fit_linear_regression(train, outcome = "mpg",
                              predictors = c("wt", "hp", "disp"),
                              cv = FALSE)
small <- fit_linear_regression(train, outcome = "mpg",
                               predictors = "wt", cv = FALSE)

res <- evaluate_regression_models(full, list(weight_only = small),
                                  newdata = test)
res
res$metrics
```

---

fit_elastic_net *Fit an elastic net, lasso or ridge regression*

Description

Fits a penalized linear model of one continuous or two-class outcome on a set of predictors, choosing how much to penalize by cross-validation, and scores the chosen model on the same folds. Which outcome it is decides which model is fitted: a numeric outcome is a linear regression and a two-class one is a logistic regression, both with the elastic net penalty on their coefficients.

Usage

```
fit_elastic_net(
  data,
  outcome,
  predictors = NULL,
  outcome_lv = NULL,
  control_label = outcome_lv[1],
  penalty = c("elastic_net", "lasso", "ridge"),
  alpha = seq(0, 1, by = 0.1),
  lambda = 10^seq(-4, 1, length.out = 50),
  cv = TRUE,
  cv_method = c("repeated_kfold", "kfold", "loocv"),
  n_fold = 5,
  n_repeat = 5,
  seed = NULL
)
```

Arguments

data	A data.frame (or matrix) in wide format, one row per observation. Typically the training half of a split_data() result.
outcome	The outcome, either the name of a column of data or a vector with one entry per row. A numeric outcome is fitted as a regression, and a factor, character or logical one as a two-class classification.
predictors	Column names to fit on, or NULL for every column of data except the outcome. Numeric, logical, factor and character columns are accepted; a column that takes a single value is left out with a message, since it cannot contribute.
outcome_lv	The two classes, reference first, so that the coefficients describe the odds of the second one. Supplying it also states that the outcome is to be classified, which is the one thing a numeric column holding two values cannot say on its own. NULL sorts the classes of an outcome that is not numeric, and leaves a numeric one to the regression.
control_label	The reference class on its own, for when the other one needs no saying. Defaults to outcome_lv[1], so a call that names one of the two names the reference either way; naming both and disagreeing is an error.
penalty	Which penalty to apply: "elastic_net" for a tuned mixture of the L1 and L2 penalties, "lasso" for the L1 penalty alone (alpha = 1), or "ridge" for the L2 penalty alone (alpha = 0). A lasso and an elastic net can set a coefficient to zero; a ridge only shrinks.
alpha	Mixing weights to search, between 0 and 1, read only when penalty = "elastic_net".
lambda	Penalty sizes to search. Larger shrinks more, and 0 is no penalty at all.
cv	Whether to cross-validate. FALSE fits the single candidate the grid names, once, and reports no resampled performance.
cv_method	Resampling scheme: "repeated_kfold" for n_repeat runs of n_fold-fold cross-validation, "kfold" for one, or "loocv" for leave-one-out.

n_fold	Folds per run, used by "repeated_kfold" and "kfold".
n_repeat	Number of runs, used by "repeated_kfold".
seed	Seed for the fold assignment, or NULL to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

The penalty is what makes this different from fit_linear_regression(). It pulls the coefficients towards zero and, when alpha is above zero, sets some of them exactly to zero, so the fit selects predictors as well as estimating them. selected in the coefficient table is that answer.

penalty names a corner of one model rather than three different models. alpha is the weight on the L1 penalty against the L2 one, so "lasso" is alpha = 1, "ridge" is alpha = 0, and "elastic_net" is everything between, alpha then being tuned like lambda. The first two ignore the alpha argument, and parameters$alpha reports what was used either way.

Cross-validation chooses the model here, and does not merely score it This is the one place where this family departs from fit_linear_regression(), where cv decides nothing about the fit. lambda and alpha are chosen by the resampled metric, which is RMSE for a regression and Accuracy for a classification, and the final model is then fitted on all usable rows at that pair. performance therefore holds one row per candidate, and parameters$alpha and parameters$lambda are the pair that won. With cv = FALSE there is nothing to choose between, so the grid must name exactly one candidate.

The estimates have no standard error, and the table says so by not having the column A penalized estimate is deliberately biased, and the usual standard error assumes an unbiased one, so there is no honest number to put under stderr, statistic, df, pval or the confidence limits. The table carries only what a penalized fit answers — estimate and selected — rather than those six columns filled with NA down their whole length, which would read as a table that lost its values instead of a model that never had them. is.null(fit$coefficients$pval) is how a consumer tells the two kinds of table apart. Use fit_linear_regression() or fit_logistic_regression() when the p-value is the point.

The predictors are penalized on a standardised scale glmnet standardises each column before penalizing it, since otherwise a predictor measured in millimetres would be shrunk less than the same predictor in metres, and returns the coefficients on their original scale. They are therefore comparable with each other in what they mean per unit, but not with an unpenalized coefficient of the same predictor, which was not shrunk at all.

Terms are not predictors As in the unpenalized models, a factor or character predictor with k levels becomes k - 1 terms named after the levels, and the coding is the one stats::lm() would have used, so the two coefficient tables can be read side by side. Each of those terms is penalized on its own, so a factor can have one level selected and another dropped.

Rows with a missing value in the outcome or in any predictor are dropped before the folds are drawn rather than inside each fit, and design$n_dropped reports how many went. A predictor that takes a single value is left out with a message, which is one of the two ways a call can end up with fewer terms than it named.

Two terms are the fewest that can be fitted. A penalty divides a budget between coefficients, and glmnet refuses a design matrix of one column outright, so a single predictor is an error here rather

than a model with nothing to trade off. fit_linear_regression() and fit_logistic_regression() fit one predictor as readily as ten.

For a two-class outcome the direction is outcome_lv, read exactly as fit_logistic_regression() reads it: the first level is the reference, every coefficient is the change in the log odds of outcome_lv[2], odds_ratio is above 1 for a predictor that raises the chance of it, and predict(model, newdata, type = "response") is its probability. That is also what the engine does unaided, glmnet modelling the last level of a factor as glm() does. control_label names the reference on its own and is enough by itself to make a column of zeroes and ones a classification; naming it alongside an outcome_lv that puts the other class first is an error, as it is in fit_logistic_regression().

New rows are predicted through the result rather than through $fit, and on this model that is the only route: glmnet was handed a design matrix and reads one by position, so the frame the fit was given cannot be handed to it again. predict.sa_model() rebuilds the terms from the levels the fit recorded and puts them in the model's order by name.

A note on $fit: caret fits the whole lambda path and records the chosen value on it, so coef() and predict() on $fit interpolate the path at that value rather than reading a model fitted at it alone. The two agree to within the resolution of the path, and everything the result reports comes from the same interpolation, so the object is consistent with itself.

Value

An object of class sa_model, the same eleven elements fit_linear_regression() returns, with these differences:

analysis "elastic_net", whichever corner penalty named.

design Holds outcome_lv, n_events and event_rate for a two-class outcome, as fit_logistic_regression() does, and neither for a continuous one.

parameters Holds penalty, the alpha and lambda that were chosen rather than the grids that were searched, and n_candidates, how many pairs were scored. The grids themselves are the rows of performance.

coefficients estimate at the chosen penalty and selected, whether the term survived it. The intercept is never penalized, so it is always selected. The inference columns are absent rather than NA; a two-class outcome also gets odds_ratio, the exponentiated estimate, but no interval to go with it.

fit_stats Measured on the rows the model was fitted to: r_squared, rmse and mae for a regression, and null_deviance, residual_deviance and mcfadden_r2 for a classification, each with n_selected and n_zero, the terms the penalty kept and dropped. These describe the fit rather than what it would do next; performance is the one that was measured on rows the model had not seen.

performance One row per candidate, the chosen one being the row that matches parameters$alpha and parameters$lambda, or NULL when cv = FALSE.

See Also

fit_linear_regression() and fit_logistic_regression() for the unpenalized fits and the inference they can report, split_data(), which defines the rows this is fitted on, coef.sa_model() for the coefficient table, predict.sa_model() for predicting the rows it was not fitted on, and coef.sa_fit() for the methods $fit answers to, among them the estimates as a named vector.

Examples

```
## A single lasso without resampling (fast enough for examples).
fit <- fit_elastic_net(mtcars, outcome = "mpg", penalty = "lasso",
                       lambda = 0.5, cv = FALSE)
fit$parameters[c("penalty", "alpha", "lambda")]
fit$coefficients[c("terms", "estimate", "selected")]


## Cross-validated grid search, ridge vs hard lasso, and known-truth scoring.
fit_elastic_net(mtcars, outcome = "mpg", penalty = "lasso",
                lambda = c(0.01, 0.1, 0.5, 1, 2),
                cv_method = "kfold", n_fold = 5, seed = 1)
hard <- fit_elastic_net(mtcars, outcome = "mpg", penalty = "lasso",
                        lambda = 2, cv = FALSE)
ridge <- fit_elastic_net(mtcars, outcome = "mpg", penalty = "ridge",
                         lambda = 2, cv = FALSE)
c(lasso = hard$fit_stats$n_selected, ridge = ridge$fit_stats$n_selected)
sim <- simulate_regression(seed = 1)
lasso <- do.call(fit_elastic_net,
                 c(sim$args, list(penalty = "lasso", lambda = 0.5,
                                  cv = FALSE)))
scored <- merge(lasso$coefficients, sim$truth_term, by = "terms")
table(planted = scored$beta != 0, selected = scored$selected)

iris2 <- iris[iris$Species != "setosa", ]
clf <- fit_elastic_net(iris2, outcome = "Species",
                       outcome_lv = c("versicolor", "virginica"),
                       penalty = "lasso", lambda = c(0.001, 0.01, 0.1),
                       cv_method = "kfold", n_fold = 5, seed = 1)
clf$coefficients[c("terms", "estimate", "odds_ratio", "selected")]
```

---

fit_linear_regression *Fit a linear regression*

Description

Fits an ordinary least squares model of one continuous outcome on a set of predictors, and scores it by cross-validation on the data it was fitted to. Both halves of that sentence matter: the coefficient table describes the fit on every usable row, while `performance` describes how the same procedure did on rows it had not seen inside each fold.

Usage

```
fit_linear_regression(
  data,
  outcome,
  predictors = NULL,
```

```
  cv = TRUE,
  cv_method = c("repeated_kfold", "kfold", "loocv"),
  n_fold = 5,
  n_repeat = 5,
  conf_level = 0.95,
  seed = NULL
)
```

Arguments

data	A data.frame (or matrix) in wide format, one row per observation. Typically the training half of a split_data() result.
outcome	The continuous outcome, either the name of a column of data or a vector with one entry per row.
predictors	Column names to fit on, or NULL for every column of data except the outcome. Numeric, logical, factor and character columns are accepted; a column that takes a single value is left out with a message, since it cannot contribute.
cv	Whether to cross-validate. FALSE fits the model once and reports no resampled performance.
cv_method	Resampling scheme: "repeated_kfold" for n_repeat runs of n_fold-fold cross-validation, "kfold" for one, or "loocv" for leave-one-out.
n_fold	Folds per run, used by "repeated_kfold" and "kfold".
n_repeat	Number of runs, used by "repeated_kfold".
conf_level	Confidence level of the coefficient intervals.
seed	Seed for the fold assignment, or NULL to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

Nothing is selected on the outcome here. predictors is the set the caller names, and a predictor that turns out not to matter stays in the table with the p-value that says so.

The model is fitted by caret::train() with method = "lm", so the fit itself is stats::lm()'s and the resampling is caret's. Two consequences are worth knowing:

Cross-validation does not change the model cv decides whether the model is scored, not how it is fitted. The final model is fitted on all usable rows either way, so the coefficients of cv = TRUE and cv = FALSE are identical and only performance and resampling differ.

Terms are not predictors A factor or character predictor with k levels becomes k - 1 terms, named after the level each one stands for. terms therefore holds the row order of coefficients, while design$predictors holds the columns that were read.

Rows with a missing value in the outcome or in any predictor are dropped before the folds are drawn rather than inside each fit. Left to the engine, deletion would happen once per fold on whatever that fold held, and the folds would then be scored on different subsets of the data; design$n_dropped reports how many rows went.

The confidence interval is the t interval on the residual degrees of freedom, the one that matches the t statistic and standard error reported beside it. A term the fit could not estimate, because another predictor already spans it, keeps its row with its estimate and inference NA and is named in a warning. Dropping the row instead would make the table quietly shorter than the model it describes.

Value

An object of class sa_model, a plain list of eleven elements.

analysis "linear_regression".

terms Coefficient term names, in the row order coefficients follows.

design What the model saw: the outcome label, its type, the predictors kept and any dropped_predictors, and the row counts n_obs, n_used and n_dropped.

parameters The fitting choices, with the resampling arguments as they were used rather than as they were passed: n_repeat is NA for a scheme that does not repeat, and both fold arguments are NA for leave-one-out.

coefficients One row per term: estimate, stderr, statistic (t), df, pval, lower_conf and upper_conf.

fit_stats The model as a whole: r_squared, adj_r_squared, sigma, the overall F test, aic and bic.

performance Resampled RMSE, Rsquared and MAE with their standard deviations across resamples, or NULL when cv = FALSE.

resampling One row per resample, or NULL.

engine What fitted the model, and which metrics it scored.

fit The caret::train() object, so that predict.sa_model() has something to predict with — with type = "response" as well as caret's own "raw", the two meaning the same thing for a regression — and so that coef(model$fit) and summary(model$fit) answer as they would for the stats::lm() inside it. This is the one element that is not portable; dropping it leaves an object that writes out as JSON.

metadata Package version, R version, platform and timestamp.

See Also

split_data(), which defines the rows this is fitted on, fit_logistic_regression() for a two-class outcome, coef.sa_model() for the coefficient table, predict.sa_model() for predicting the rows it was not fitted on, and coef.sa_fit() for the methods $fit answers to, among them the estimates as a named vector.

Examples

```
## Fitted without resampling (fast enough for examples).
fit <- fit_linear_regression(mtcars, outcome = "mpg",
                             predictors = c("wt", "hp", "disp"),
                             cv = FALSE)
fit$coefficients
```

```
## Cross-validation scores the model; it does not change the coefficients.
scored <- fit_linear_regression(mtcars, outcome = "mpg",
                                predictors = c("wt", "hp", "disp"),
                                cv_method = "kfold", seed = 1)
all.equal(fit$coefficients, scored$coefficients)


## A factor predictor becomes one term per level beyond the first.
cars <- mtcars
cars$cyl <- factor(cars$cyl)
fit_linear_regression(cars, outcome = "mpg", predictors = c("wt", "cyl"),
                      cv = FALSE)$terms

## The training half of a split is what a model is normally fitted on.
sp <- split_data(mtcars, stratified = "mpg", seed = 1)
train <- sp$datasets[[1]]$train_data
fit_linear_regression(train, outcome = "mpg", predictors = c("wt", "hp"),
                      cv = FALSE)$fit_stats$r_squared
```

`fit_logistic_regression` *Fit a logistic regression*

Description

Fits a binomial logistic regression of one two-class outcome on a set of predictors, and scores it by cross-validation on the data it was fitted to. The coefficient table describes the fit on every usable row, while `performance` describes how the same procedure did on rows it had not seen inside each fold.

Usage

```
fit_logistic_regression(
  data,
  outcome,
  predictors = NULL,
  outcome_lv = NULL,
  control_label = outcome_lv[1],
  cv = TRUE,
  cv_method = c("repeated_kfold", "kfold", "loocv"),
  n_fold = 5,
  n_repeat = 5,
  conf_level = 0.95,
  seed = NULL
)
```

Arguments

data	A data.frame (or matrix) in wide format, one row per observation. Typically the training half of a split_data() result.
outcome	The two-class outcome, either the name of a column of data or a vector with one entry per row. Factor, character, logical and numeric columns are all read as class labels.
predictors	Column names to fit on, or NULL for every column of data except the outcome. Numeric, logical, factor and character columns are accepted; a column that takes a single value is left out with a message, since it cannot contribute.
outcome_lv	The two classes, reference first, so that the coefficients describe the odds of the second one. NULL sorts the classes, which puts "control" before "treated" and 0 before 1.
control_label	The reference class on its own, for when the other one needs no saying. Defaults to outcome_lv[1], so a call that names one of the two names the reference either way; naming both and disagreeing is an error.
cv	Whether to cross-validate. FALSE fits the model once and reports no resampled performance.
cv_method	Resampling scheme: "repeated_kfold" for n_repeat runs of n_fold-fold cross-validation, "kfold" for one, or "loocv" for leave-one-out.
n_fold	Folds per run, used by "repeated_kfold" and "kfold".
n_repeat	Number of runs, used by "repeated_kfold".
conf_level	Confidence level of the coefficient intervals.
seed	Seed for the fold assignment, or NULL to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

outcome_lv fixes the direction, and it does so by the rule the rest of the package follows: the first level is the reference. Every coefficient is the change in the log odds of outcome_lv[2] per unit of its predictor, and odds_ratio is above 1 for a predictor that raises the chance of it. With outcome_lv = c("control", "case") the table therefore reads as a statement about case, and the same vector handed to compare_two_groups() as group_lv would put control in the denominator of its fold change, so the two point the same way.

control_label states the same direction with one name instead of two, which is the shorter thing to say when the other class needs no saying. Naming both and pointing them at different classes is an error rather than a re-pointing: outcome_lv holds the two classes and nothing else, so there is no reading under which a different reference leaves any of it standing. The comparison functions take the argument the other way round, since their group_lv carries the display order too — see compare_two_groups().

This is also what the engine does unaided. stats::glm() models the probability of the last level of a factor, so ordering the levels reference-first is the whole of the implementation and nothing is reversed afterwards.

The rule reaches the predictions too: predict(model, newdata, type = "response") is the probability of outcome_lv[2], the same class the coefficients describe. See predict.sa_model(), and

coef.sa_fit() for the same word on the engine object, since "response" is one this package adds to the ones caret::train() accepts.

A third class is an error rather than a silently dropped set of rows: two classes are what this model is, and quietly fitting a subset of the data that was passed in would answer a question nobody asked. Reduce data to the two classes first; naming two of three with outcome_lv is refused for the same reason.

The rest matches fit_linear_regression(): the model is fitted by caret::train() with method = "glm", cv decides whether the model is scored rather than how it is fitted, a factor predictor with k levels becomes k - 1 terms, and rows missing anything the model needs are dropped before the folds are drawn. The interval is the Wald interval, the one matching the z statistic and standard error reported beside it, rather than the profile likelihood interval stats::confint() would give.

Perfect separation is reported rather than hidden. A predictor that splits the two classes exactly gives an estimate that grows until the fit stops, with a standard error to match, and the engine says so once per fold; those notes come back as one message with a count.

Value

An object of class sa_model, the same eleven elements fit_linear_regression() returns, with these differences:

analysis "logistic_regression".

design Also holds outcome_lv, and n_events with event_rate, the number and proportion of rows in outcome_lv[2].

coefficients statistic is a Wald z rather than a t and df is NA, since the z is not referred to any. odds_ratio, or_lower_conf and or_upper_conf are added, being the exponentiated estimate and its limits.

fit_stats null_deviance and residual_deviance with their degrees of freedom, mcfadden_r2, the likelihood ratio test of the model against the intercept alone, aic and bic.

performance Resampled Accuracy and Kappa rather than the regression metrics.

See Also

split_data(), which defines the rows this is fitted on, fit_linear_regression() for a continuous outcome, coef.sa_model() for the coefficient table, predict.sa_model() for predicting the rows it was not fitted on, and coef.sa_fit() for the methods $fit answers to, among them the estimates as a named vector.

Examples

```
## Two of the three iris species, so that the outcome has two classes.
## `versicolor` first makes it the reference, so every odds ratio is the odds
## of `virginica` rather than of the other way round.
iris2 <- iris[iris$Species != "setosa", ]
fit <- fit_logistic_regression(iris2, outcome = "Species",
                               predictors = c("Petal.Length", "Sepal.Width"),
                               outcome_lv = c("versicolor", "virginica"),
                               cv = FALSE)
fit$coefficients[c("terms", "estimate", "odds_ratio", "pval")]
```

```
fit$fit_stats$mcfadden_r2

## Swapping the two levels turns every coefficient around and inverts every
## odds ratio, which is the same rule `group_lv` follows in a comparison.
other_way <- fit_logistic_regression(
  iris2, outcome = "Species", predictors = c("Petal.Length", "Sepal.Width"),
  outcome_lv = c("virginica", "versicolor"), cv = FALSE
)
cbind(versicolor_ref = fit$coefficients$odds_ratio,
      virginica_ref  = other_way$coefficients$odds_ratio)

## The training half of a split is what a model is normally fitted on, and
## stratifying on the outcome keeps both halves able to see both classes.
sp <- split_data(iris2, stratified = "Species", seed = 1)
train <- sp$datasets[[1]]$train_data
fit_logistic_regression(train, outcome = "Species",
                        predictors = "Petal.Width", cv = FALSE)$design$n_events
```

`fit_rf` *Fit a random forest*

Description

Grows a forest of regression or classification trees on a set of predictors, and scores it by cross-validation on the data it was fitted to. Which outcome it is decides which forest is grown: a numeric outcome is a regression forest and a two-class one a classification forest, as in `fit_elastic_net()`, so one function covers what `fit_linear_regression()` and `fit_logistic_regression()` cover between them.

Usage

```
fit_rf(
  data,
  outcome,
  predictors = NULL,
  outcome_lv = NULL,
  control_label = outcome_lv[1],
  mtry = NULL,
  ntree = 500,
  nodesize = NULL,
  cv = TRUE,
  cv_method = c("repeated_kfold", "kfold", "loocv"),
  n_fold = 5,
  n_repeat = 5,
  seed = NULL
)
```

Arguments

`data`	A data.frame (or matrix) in wide format, one row per observation. Typically the training half of a `split_data()` result.
`outcome`	The outcome, either the name of a column of `data` or a vector with one entry per row. A numeric outcome is fitted as a regression, and a factor, character or logical one as a two-class classification.
`predictors`	Column names to fit on, or `NULL` for every column of `data` except the outcome. Numeric, logical, factor and character columns are accepted; a column that takes a single value is left out with a message, since it cannot contribute.
`outcome_lv`	The two classes, reference first, so that the reported sensitivity and the predicted probability are about the second one. Supplying it also states that the outcome is to be classified, which is the one thing a numeric column holding two values cannot say on its own. `NULL` sorts the classes of an outcome that is not numeric, and leaves a numeric one to the regression.
`control_label`	The reference class on its own, for when the other one needs no saying. Defaults to `outcome_lv[1]`, so a call that names one of the two names the reference either way; naming both and disagreeing is an error.
`mtry`	Predictors offered at each split, one value to score or several to choose between. `NULL` is the rule of thumb: `floor(sqrt(p))` for a classification and `floor(p / 3)` for a regression, at least 1. A value above the predictor count is refused rather than quietly reset by the engine.
`ntree`	Trees to grow. More is steadier rather than more prone to overfitting, and costs time.
`nodesize`	Smallest number of rows a leaf may hold, so larger grows shallower trees. `NULL` is `randomForest`'s own default: 1 for a classification and 5 for a regression.
`cv`	Whether to cross-validate. `FALSE` grows the single candidate the grid names, once, and reports no resampled performance. The out-of-bag `fit_stats` are there either way.
`cv_method`	Resampling scheme: `"repeated_kfold"` for `n_repeat` runs of `n_fold`-fold cross-validation, `"kfold"` for one, or `"loocv"` for leave-one-out.
`n_fold`	Folds per run, used by `"repeated_kfold"` and `"kfold"`.
`n_repeat`	Number of runs, used by `"repeated_kfold"`.
`seed`	Seed for the forest and the fold assignment, or `NULL` to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

What comes back in place of a coefficient table is an importance table: one row per predictor, in descending order of how much the forest lost when that predictor was shuffled. It says which predictors carried the fit and not which way they pushed it, since a forest is free to use the same predictor in opposite directions in different regions of the data.

`estimate` **is permutation importance, and there is no standard error beside it** Each tree is scored on its out-of-bag rows, then scored again with one predictor's values shuffled among them, and

`estimate` is the mean loss over the trees: `%IncMSE` for a regression and `MeanDecreaseAccuracy` for a classification. `impurity` is the other measure the same fit reports, the total drop in node impurity that predictor was responsible for — `IncNodePurity` or `MeanDecreaseGini`. Neither is an estimate of anything a distribution is defined over, so the inference columns are absent from the table rather than present and `NA`, exactly as they are for `fit_elastic_net()`, and `is.null(fit$coefficients$pval)` tells the two kinds of table apart. The values are the unscaled ones. `randomForest::importance()` divides them by their standard deviation across trees by default, which makes a t-shaped ratio that is not referred to any distribution, so what is reported is the mean loss itself, on the scale of the metric it was measured in.

A negative importance is a value, not a gap Shuffling a predictor that carried nothing can leave the forest very slightly better than it was, and the mean loss then comes out below zero. It reads as it should: this predictor did no better than its own permutation.

The terms are the predictors A tree splits a factor on its levels directly, so nothing is dummy coded and a k-level factor is one term rather than `k - 1`. This is the one model here where `terms` and `design$predictors` hold the same names — in a different order, since the table is sorted by importance and the design records the order the columns were read in.

`fit_stats` **is measured out of bag** Every tree is grown on a bootstrap sample of the rows, which leaves the rest of them out of bag for that tree, and `randomForest` predicts each row from the trees that did not see it. That is already a held-out score, and it is the honest one: a forest asked to predict the rows it was fitted to reports something close to perfect whatever the data held. The `oob_` prefix says these numbers are not the in-sample kind the other models report.

Cross-validation chooses `mtry` **when there is more than one to choose from** `mtry` is the only argument `caret::train()` tunes for a forest; `ntree` and `nodesize` are the same for every candidate. A single value is scored rather than chosen, as in `fit_linear_regression()`, and a vector is compared the way `fit_elastic_net()` compares penalties, `parameters$mtry` then holding the value that won and `performance` one row per candidate. Unlike a penalty, `mtry` has a default worth fitting, so `NULL` resolves to the rule of thumb — the square root of the predictor count for a classification, a third of it for a regression — rather than to a grid, and `cv = FALSE` needs nothing added to work.

Rows with a missing value in the outcome or in any predictor are dropped before the folds are drawn rather than inside each fit, and `design$n_dropped` reports how many went. A predictor that takes a single value is left out with a message.

One predictor is enough. There is no budget to divide as there is under a penalty, so `predictors = "wt"` grows a forest of stumps on that one column rather than raising the error `fit_elastic_net()` raises.

For a two-class outcome the direction is `outcome_lv`, read as `fit_logistic_regression()` reads it: the first level is the reference, so `oob_sensitivity` is the share of `outcome_lv[2]` the forest found, `oob_specificity` the share of `outcome_lv[1]` it left alone, and `predict(model, newdata, type = "response")` is the probability of `outcome_lv[2]`. The importance table itself has no direction to report, which is why there is no `odds_ratio` column beside it. `control_label` says the same thing with one name instead of two and says it alone, so it too is enough to make a column of zeroes and ones a classification. Naming both and pointing them at different classes is an error rather than a re-pointing, since an `outcome_lv` holds the two classes and nothing else; the comparison functions read the argument the other way, as `compare_two_groups()` describes.

A forest is random beyond the folds — the rows of each tree and the predictors of each split are both draws — so `seed` fixes more here than it does in the other models, where it fixes only the fold

assignment. Two calls without one give slightly different importance values on the same data.

Value

An object of class `sa_model`, the same eleven elements `fit_linear_regression()` returns, with these differences:

`analysis` `"random_forest"`.

`terms` The predictors, in descending order of importance, since a forest expands no factor into dummy terms.

`design` Holds `outcome_lv`, `n_events` and `event_rate` for a two-class outcome, as `fit_logistic_regression()` does, and neither for a continuous one.

`parameters` Holds the `mtry` that was fitted rather than the grid that was searched, `ntree`, `nodesize`, and `n_candidates`, how many values of `mtry` were scored. The grid itself is the rows of `performance`.

`coefficients` `estimate`, the permutation importance, and `impurity`, the impurity-based one. There is no intercept row, no `odds_ratio` and no inference column.

`fit_stats` Measured on the out-of-bag predictions: `oob_r_squared`, `oob_rmse` and `oob_mae` for a regression, and `oob_accuracy`, `oob_error`, `oob_kappa`, `oob_sensitivity` and `oob_specificity` for a classification, each with `n_oob`, the rows that were out of bag of at least one tree and so could be predicted.

`performance` One row per value of `mtry`, the chosen one being the row that matches `parameters$mtry`, or `NULL` when `cv = FALSE`.

See Also

`fit_linear_regression()`, `fit_logistic_regression()` and `fit_elastic_net()` for the models that do report an effect per predictor, `split_data()`, which defines the rows this is fitted on, `coef.sa_model()` for the importance table, and `predict.sa_model()` for predicting the rows it was not fitted on.

Examples

```
## A single forest without resampling (fast enough for examples).
fit <- fit_rf(mtcars, outcome = "mpg", ntree = 50, cv = FALSE, seed = 1)
fit$coefficients
fit$fit_stats


## Cross-validated fit and an `mtry` search.
fit_rf(mtcars, outcome = "mpg", ntree = 200,
       cv_method = "kfold", n_fold = 5, seed = 1)
tuned <- fit_rf(mtcars, outcome = "mpg", mtry = c(2, 5, 10), ntree = 200,
                cv_method = "kfold", n_fold = 5, seed = 1)
tuned$parameters[c("mtry", "n_candidates")]

## Two-class outcome and a simulated regression scored against known truth.
iris2 <- iris[iris$Species != "setosa", ]
clf <- fit_rf(iris2, outcome = "Species",
```

```
              outcome_lv = c("versicolor", "virginica"),
              ntree = 200, cv = FALSE)
clf$fit_stats[c("oob_accuracy", "oob_sensitivity", "oob_specificity")]
sim <- simulate_regression(seed = 1)
rf <- do.call(fit_rf, c(sim$args, list(ntree = 200, cv = FALSE)))
scored <- merge(rf$coefficients, sim$truth_term, by = "terms")
tapply(scored$estimate, scored$beta != 0, mean)
```

`fit_svm` *Fit a support vector machine*

Description

Fits a support vector machine with a radial basis kernel on a set of predictors, choosing the cost and the kernel width by cross-validation, and scores the chosen machine on the same folds. Which outcome it is decides which machine is fitted: a numeric outcome is a support vector regression and a two-class one a classification, as in `fit_elastic_net()` and `fit_rf()`, so one function covers what `fit_linear_regression()` and `fit_logistic_regression()` cover between them.

Usage

```
fit_svm(
  data,
  outcome,
  predictors = NULL,
  outcome_lv = NULL,
  control_label = outcome_lv[1],
  C = 2^seq(-5, 10, by = 2),
  sigma = NULL,
  n_permute = 10,
  cv = TRUE,
  cv_method = c("repeated_kfold", "kfold", "loocv"),
  n_fold = 5,
  n_repeat = 5,
  seed = NULL
)
```

Arguments

`data`	A data.frame (or matrix) in wide format, one row per observation. Typically the training half of a `split_data()` result.
`outcome`	The outcome, either the name of a column of `data` or a vector with one entry per row. A numeric outcome is fitted as a regression, and a factor, character or logical one as a two-class classification.

`predictors` Column names to fit on, or `NULL` for every column of `data` except the outcome. Numeric, logical, factor and character columns are accepted; a column that takes a single value is left out with a message, since it cannot contribute.

`outcome_lv` The two classes, reference first, so that the reported sensitivity and the predicted probability are about the second one. Supplying it also states that the outcome is to be classified, which is the one thing a numeric column holding two values cannot say on its own. `NULL` sorts the classes of an outcome that is not numeric, and leaves a numeric one to the regression.

`control_label` The reference class on its own, for when the other one needs no saying. Defaults to `outcome_lv[1]`, so a call that names one of the two names the reference either way; naming both and disagreeing is an error.

`C` Cost of violating the margin, one value to score or several to choose between. Larger fits the training rows harder and generalises less.

`sigma` Width of the radial kernel on the standardised scale, one value or several. Larger makes the kernel narrower, so each support vector reaches fewer rows. `NULL` is the middle of the three widths `kernlab::sigest()` reports for these rows.

`n_permute` Shuffles per term behind each importance. More is steadier and costs one prediction of every row per shuffle.

`cv` Whether to cross-validate. `FALSE` fits the single candidate the grid names, once, and reports no resampled performance.

`cv_method` Resampling scheme: `"repeated_kfold"` for `n_repeat` runs of `n_fold`-fold cross-validation, `"kfold"` for one, or `"loocv"` for leave-one-out.

`n_fold` Folds per run, used by `"repeated_kfold"` and `"kfold"`.

`n_repeat` Number of runs, used by `"repeated_kfold"`.

`seed` Seed for the kernel width, the class probabilities, the shuffles behind the importance and the fold assignment, or `NULL` to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

What comes back in place of a coefficient table is an importance table: one row per term, in descending order of how much the machine lost when that term's values were shuffled among the rows. It says which terms carried the fit and not which way they pushed it, since a kernel machine bends its surface in different directions in different regions of the data.

`estimate` **is permutation importance, and there is no standard error beside it** The fitted machine is scored on the rows it was fitted to, then scored again with one term's values shuffled among those rows, and `estimate` is the mean loss over `n_permute` shuffles. The metric is the one the resampling tuned on: the rise in `RMSE` for a regression and the fall in `Accuracy` for a classification, so a larger value is a term the machine needed either way. It is not an estimate of anything a distribution is defined over, so the inference columns are absent from the table rather than present and `NA`, exactly as they are for `fit_elastic_net()` and `fit_rf()`, and `is.null(fit$coefficients$pval)` tells the two kinds of table apart.

The importance is measured in sample, and a forest's is not This is the one number here that is weaker than `fit_rf()`'s counterpart. Every tree of a forest leaves about a third of the rows out of its bootstrap sample, so the permutation can be scored on rows that tree had not seen; a machine is fitted on all of them at once and has no such rows. A term the machine fitted noise with therefore earns some importance here that it would not earn on held-out rows. `performance` is the number that was measured on rows the procedure had not seen, and it describes the whole machine rather than one term of it.

A negative importance is a value, not a gap Shuffling a term that carried nothing can leave the machine very slightly better than it was, and the mean loss then comes out below zero. It reads as it should: this term did no better than its own permutation.

Terms are not predictors A factor or character predictor with `k` levels becomes `k - 1` terms named after the levels, and the coding is the one `stats::lm()` would have used, so the tables can be read side by side. Each of those terms is shuffled on its own, so one level of a factor can matter to the machine and another not. `fit_rf()` is the model where the terms are the predictors themselves, because a tree splits a factor directly.

The predictors are centred and scaled A radial kernel reads one distance between two rows, so a predictor measured in millimetres would dominate the same predictor in metres and the kernel width would mean something different along each axis. Every term is centred and scaled before the machine sees it, and `predict.sa_model()` applies the same centre and scale to new rows, so this is not something to undo afterwards. It is also why `sigma` is a width on the standardised scale rather than on the scale the columns arrived in.

Cross-validation chooses `C` and `sigma` Both are chosen by the resampled metric, which is `RMSE` for a regression and `Accuracy` for a classification, and the final machine is then fitted on all usable rows at the pair that won, as in `fit_elastic_net()`. `performance` therefore holds one row per candidate and `parameters` the pair that was fitted. With `cv = FALSE` there is nothing to choose between, so `C` and `sigma` must name exactly one candidate.

`sigma = NULL` is a width read off the data rather than a grid, since the distances between the rows are what fixes the scale a kernel is comparing them on. `kernlab::sigest()` reports three widths, from the 10th, 50th and 90th percentile of those distances, and the middle one is used. It samples pairs of rows to do it, so this is one of the things `seed` fixes.

Rows with a missing value in the outcome or in any predictor are dropped before the folds are drawn rather than inside each fit, and `design$n_dropped` reports how many went. A predictor that takes a single value is left out with a message.

One predictor is enough. A distance can be measured along one axis as readily as along ten, so a single column is a machine here rather than the error `fit_elastic_net()` raises for want of a budget to divide.

For a two-class outcome the direction is `outcome_lv`, read as `fit_logistic_regression()` reads it: the first level is the reference, so `sensitivity` is the share of `outcome_lv[2]` the machine found, `specificity` the share of `outcome_lv[1]` it left alone, and `predict(model, newdata, type = "response")` is the probability of `outcome_lv[2]`. Those probabilities are Platt's, a logistic curve fitted to the decision values by an internal cross-validation of `kernlab`'s own, which is a second thing `seed` fixes and a reason two machines fitted without one predict slightly different probabilities from the same decisions. The importance table has no direction to report, which is why there is no `odds_ratio` column beside it. `control_label` names the reference on its own and is enough by itself to make a column of zeroes and ones a classification; naming it alongside an `outcome_lv` that puts the other class first is an error, as it is in `fit_logistic_regression()`.

Value

An object of class `sa_model`, the same eleven elements `fit_linear_regression()` returns, with these differences:

`analysis` `"svm"`.

`terms` The model terms, in descending order of importance.

`design` Holds `outcome_lv`, `n_events` and `event_rate` for a two-class outcome, as `fit_logistic_regression()` does, and neither for a continuous one.

`parameters` Holds `kernel`, the `C` and `sigma` that were fitted rather than the grids that were searched, `n_candidates`, how many pairs were scored, and `n_permute`. The grids themselves are the rows of `performance`.

`coefficients` `estimate`, the permutation importance. There is no intercept row, no `odds_ratio` and no inference column.

`fit_stats` Measured on the rows the machine was fitted to: `r_squared`, `rmse` and `mae` for a regression, and `accuracy`, `error`, `kappa`, `sensitivity` and `specificity` for a classification, each with `n_support_vector` and `support_vector_rate`, the rows the machine kept as support vectors and their share of the rows it was fitted to. These describe the fit rather than what it would do next; `performance` is the one that was measured on rows the machine had not seen.

`performance` One row per candidate, the chosen one being the row that matches `parameters$C` and `parameters$sigma`, or `NULL` when `cv = FALSE`.

See Also

`fit_rf()` for the other model that answers with importance rather than with coefficients, `fit_linear_regression()`, `fit_logistic_regression()` and `fit_elastic_net()` for the models that do report an effect per predictor, `split_data()`, which defines the rows this is fitted on, `coef.sa_model()` for the importance table, and `predict.sa_model()` for predicting the rows it was not fitted on.

Examples

```
## A single machine without resampling (fast enough for examples).
fit <- fit_svm(mtcars, outcome = "mpg", predictors = c("wt", "hp", "disp"),
               C = 1, cv = FALSE, seed = 1)
fit$coefficients
fit$fit_stats


## Cross-validated fit and a search over `C`.
fit_svm(mtcars, outcome = "mpg", predictors = c("wt", "hp", "disp"),
        C = 1, cv_method = "kfold", n_fold = 5, seed = 1)
tuned <- fit_svm(mtcars, outcome = "mpg", predictors = c("wt", "hp", "disp"),
                 C = c(0.5, 2, 8), cv_method = "kfold", n_fold = 5, seed = 1)
tuned$parameters[c("C", "sigma", "n_candidates")]

## Two-class outcome and a simulated regression scored against known truth.
iris2 <- iris[iris$Species != "setosa", ]
clf <- fit_svm(iris2, outcome = "Species",
```

```
                outcome_lv = c("versicolor", "virginica"), C = 1, cv = FALSE,
                seed = 1)
clf$fit_stats[c("accuracy", "sensitivity", "specificity")]
sim <- simulate_regression(seed = 1)
svm <- do.call(fit_svm, c(sim$args, list(C = 1, cv = FALSE, seed = 1)))
scored <- merge(svm$coefficients, sim$truth_term, by = "terms")
tapply(scored$estimate, scored$beta != 0, mean)
```

`make_block_cor` *Build a block correlation matrix*

Description

Assembles a correlation matrix out of groups of predictors that correlate with each other, which is the structure worth simulating when the question is how a model behaves under collinearity. Every predictor inside a block correlates with every other one in it at the same value, and everything outside every block correlates at `default_cor`. A block that names `against` is split in two instead: each side agrees within itself and disagrees with the other side.

Usage

```
make_block_cor(n_features, blocks = list(), default_cor = 0)
```

Arguments

`n_features`	Number of predictors the matrix describes, so its size.
`blocks`	List of blocks, each a list with `features`, the indices in the block, and `cor`, the correlation they share. A block may also name `against`, further indices that correlate at `cor` among themselves and at `-cor` with the ones in `features`. `list()` gives a matrix with `default_cor` everywhere off the diagonal.
`default_cor`	Correlation between any two predictors that are not in a block together. The default of `0` leaves them independent.

Details

The matrix is what `simulate_regression()` and `simulate_classification()` take as `cor_mat`. Its point there is that a null predictor correlated with a planted one is not distinguishable from the planted one by the data alone, so its estimated coefficient is drawn away from the zero it really has. That is the single most common reason a coefficient table names the wrong predictor, and it cannot be shown at all with independent predictors.

Each block is its own `list()`. A single `list()` that names `features` twice is one block and not two, because `$` reads the first of a repeated name and nothing else, so the second pair of values would be dropped without a word. Repeated and unknown names in a block are refused for that reason.

Blocks may not overlap. A predictor in two blocks would have two correlations with the same partner and only one of them could be written down, so the second block would silently win. Nest the smaller correlation as `default_cor` and name one block instead, or, when the second group is what the first moves against, name it as `against` in one block.

How strong a negative correlation one block can hold depends on how many predictors are in it. A block of `k` predictors sharing one value is positive definite only above `-1/(k - 1)`: two predictors may disagree at -0.9, three at no more than -0.5, four at no more than -0.333. Three predictors cannot all disagree strongly, since whichever way the third moves it agrees with one of the first two.

`against` is how a strong negative correlation is written down instead. `list(features = 1:3, cor = 0.9, against = 4:6)` puts 0.9 among the first three, 0.9 among the last three and -0.9 between the two sides. Splitting a block by which way its predictors move leaves it positive definite for any `cor` below 1 whatever its size, since it is then one factor with a sign per predictor rather than a demand that everything disagree at once.

The result is checked for positive definiteness, which is the property that separates a matrix of correlations from a matrix of numbers between -1 and 1. The blocks and `default_cor` are checked one at a time first, so that a value no block of that size could hold is named as such, and the eigenvalue of the assembled matrix is what reports a `default_cor` the blocks cannot sit inside.

Value

A symmetric `n_features` by `n_features` matrix with 1 on the diagonal.

See Also

`simulate_regression()` and `simulate_classification()`, which take the result as `cor_mat`.

Examples

```
## Six predictors: the first two nearly interchangeable, the next three
## moderately related, and the sixth on its own.
cor_mat <- make_block_cor(
  n_features = 6,
  blocks = list(
    list(features = 1:2, cor = 0.8),
    list(features = 3:5, cor = 0.5)
  )
)
round(cor_mat, 2)

## Handed to a simulator, it is what makes a null predictor look like a
## planted one. `x_2` has a coefficient of exactly zero and correlates 0.8
## with a predictor that does not.
sim <- simulate_regression(n_pred = 6, beta = c(2, 0, 0, 0, 0, 0),
                           cor_mat = cor_mat, seed = 1)
sim$truth[1:2, c("predictors", "role", "beta", "max_cor_signal")]

## Three predictors moving one way and three the other. `against` carries the
## sign, so -0.9 between the sides is available at any block size.
opposed <- make_block_cor(
  n_features = 6,
```

```
  blocks = list(list(features = 1:3, cor = 0.9, against = 4:6))
)
round(opposed, 2)

## Three predictors cannot all disagree at -0.6, and the limit for a block of
## three is named rather than left to the matrix.
try(make_block_cor(6, list(list(features = 1:3, cor = -0.6))))

## Each block needs its own `list()`. One `list()` naming `features` twice is
## a single block whose second value R would never read.
try(make_block_cor(6, list(list(features = 1:3, cor = 0.9,
                                features = 4:6, cor = -0.4))))

## Four predictors cannot all disagree with each other, so a request for it is
## refused here rather than met later by a draw that quietly ignores it.
try(make_block_cor(4, default_cor = -0.5))
```

`perform_pca` *Reduce many features to a few components*

Description

Rotates a wide table of samples and features onto the axes that carry the most variance, so that a few coordinates stand in for all of them. The components are ordered by how much of the variance they carry, `$variance` says how much that is, and `$loadings` says which features built each one.

Usage

```
perform_pca(
  data,
  feats = NULL,
  embedding_scale = c("samples", "features"),
  center = TRUE,
  scale = TRUE
)
```

Arguments

`data`	A data.frame or a matrix in wide format, one row per sample and one column per feature. This is the same layout `compare_two_groups()` and `draw_heatmap()` take. Row names are kept as the sample labels, repeated ones included, since a sample name is a naming choice rather than a key; rows without a name are labelled by position.
`feats`	Column names to reduce, or `NULL` for every numeric column of `data`. A non-numeric column is left out with a message, so a frame that carries a grouping column alongside the measurements can be passed as it is.

`embedding_scale`
: Which margin becomes the points of the picture: `"samples"`, the default, for one point per row of `data`, or `"features"` for one point per column. `design$point_type` reports it. See the details: this is not the same as transposing `data` yourself.

`center`, `scale`
: Whether to centre each feature and divide it by its standard deviation before rotating. Scaling is on by default because features are not measured on a common scale, and without it the feature with the widest units decides where the first component points. Both always apply to the **columns of** `data`, whatever `embedding_scale` is.

Details

The input is the wide format the comparison functions take: **one row per sample and one column per feature**. What comes back has one row per point, in one order every table follows, which is what makes a reduction plottable against anything else read from the same frame.

Value

An object of class `sa_reduction`, a plain list.

`analysis` `"pca"`.

`points` Labels of the things that became points — samples or features, as `embedding_scale` asked — in the row order `scores` follows. This is what `features` is to a comparison result.

`design` What was reduced: `point_type`, either `"sample"` or `"feature"`, the `feats` kept and any `dropped_feats`, and the counts `n_samples`, `n_used`, `n_dropped` and `n_feats`. The counts describe the input rather than the picture, so `n_samples` is always rows of `data` and `n_feats` always kept columns, whichever margin became the points.

`parameters` `embedding_scale` as resolved, and the two flags.

`variance` One row per component: `component`, `sdev`, `prop_var` as a percentage and `cum_var`. Every component is here, since a share of the variance is only a share if the rest of it is there to be a share of.

`loadings` The margin that was not embedded: one row per variable, `variables` beside one column per component.

`scores` Coordinates: `points` beside one column per component.

`engine` What computed the rotation.

`fit` The `stats::prcomp()` object, always fitted to the samples. This is the slot that is not portable; dropping it leaves an object that writes out as JSON.

`metadata` Package version, R version, platform and timestamp.

Which slot holds what

`$fit` is the `stats::prcomp()` object itself, always fitted to the samples, since that is the fit both margins are read from. `$scores` is the margin `embedding_scale` asked for and is what a scatter plot of the points is drawn from. On the default sample scale the two agree: `$scores` is `$fit$x` beside a `points` column, and `$loadings` is `$fit$rotation`.

Choosing the margin, rather than transposing

`embedding_scale = "features"` is how a map of the features is asked for, and hand-transposing the input is not the same thing. `stats::prcomp()` centres and scales **columns**: on a sample-by-feature matrix that standardises features, which is what this method needs, and on the transpose it standardises samples instead. So `perform_pca(t(data))` runs a third analysis that is neither of the two on offer. On the 8-feature simulation in the examples its first component correlates 0.82 with the loadings, where `embedding_scale = "features"` correlates 1.

What the argument does instead is leave the matrix alone. One decomposition already answers on both margins, so the features are the rows of `$rotation` and nothing has to be recomputed. What `$scores` reports is those loadings rescaled to variance-weighted length, `$rotation %*% diag(sdev * sqrt(n - 1))`: the same map, on the scale a coordinate has rather than the unit length a direction has. `$loadings` is then the margin that was not embedded, one row per sample, and `$variance` does not change at all, so the axis labels a plot carries are the same on both scales.

Transposing by hand is right only when the rows of `data` really are features, which is how an expression matrix usually arrives. Then the transpose puts the data into this function's layout and `embedding_scale` chooses from there.

What is dropped before anything runs

`stats::prcomp()` does not accept a missing value, so rows that are not complete and finite across `feats` are dropped before it is called, and `design$n_dropped` reports how many went. This is the listwise deletion the rest of the package uses; nothing is imputed.

A feature that takes a single value cannot be scaled — the division is by zero — so with `scale = TRUE` it is left out with a message and named in `design$dropped_feats`. With `scale = FALSE` it stays and becomes a component of no variance.

Portability

Everything but `$fit` is a data.frame, a character vector or a named list, so dropping that one slot leaves an object that writes out as JSON. In a Python transcription this is `sklearn.decomposition.PCA`.

See Also

`perform_tsne()` and `perform_umap()`, which answer the same question in a way that can follow structure a rotation cannot, and `draw_heatmap()`, which shows the same wide input a cell at a time instead of a point at a time.

Examples

```
## The sample coordinates are `$scores`, and `$variance` is what labels the axes.
res <- perform_pca(iris[1:4])
res
head(res$scores)
res$variance

plot(res$scores[c("PC1", "PC2")],
     xlab = paste0("PC1 (", round(res$variance$prop_var[1], 2), "%)"),
     ylab = paste0("PC2 (", round(res$variance$prop_var[2], 2), "%)"),
     col = as.integer(iris$Species), pch = 16)
```

```
## `$loadings` is the other axis: which features built each component.
res$loadings

## On data with a planted two-group structure. The group was never shown to the
## rotation, so the split is its to find.
sim <- simulate_two_groups(n_feats = 30, n_up = 5, n_down = 5, seed = 3)
by_samp <- perform_pca(sim$args$data)
table(group = sim$args$group, side = by_samp$scores$PC1 > 0)

## `embedding_scale = "features"` turns the features into the points. Here the
## correlation blocks are planted, and the rotation was not told about them.
cor_mat <- make_block_cor(
  n_features = 8,
  blocks = list(list(features = 1:2, cor = 0.8),
                list(features = 3:5, cor = 0.5),
                list(features = 7:8, cor = 0.9))
)
blocks <- simulate_classification(cor_mat = cor_mat, seed = 2026)$args$data
by_feat <- perform_pca(blocks, feats = paste0("x_", 1:8),
                       embedding_scale = "features")
by_feat
by_feat$points

plot(by_feat$scores[c("PC1", "PC2")], type = "n",
     xlab = paste0("PC1 (", round(by_feat$variance$prop_var[1], 2), "%)"),
     ylab = paste0("PC2 (", round(by_feat$variance$prop_var[2], 2), "%)"))
text(by_feat$scores[c("PC1", "PC2")], labels = by_feat$points, font = 2)

## The same fit read from the other end: the loadings of the sample scale,
## rescaled from unit length to variance-weighted length, on the same axis
## proportions.
same <- perform_pca(blocks, feats = paste0("x_", 1:8))
all.equal(by_feat$variance, same$variance)
round(diag(cor(by_feat$scores[-1], same$loadings[-1])), 6)
```

`perform_rfe` *Select the predictors worth keeping*

Description

Runs a recursive feature elimination: the predictors are ranked, the weakest is dropped, and the model is scored again, over and over, so that every subset size gets a resampled score. What comes back is the size that scored best, the predictors of that size, and the whole profile the search walked, so that a choice of two predictors over eight can be read against what the other six were worth.

Usage

```
perform_rfe(
```

```
  data,
  outcome,
  predictors = NULL,
  outcome_lv = NULL,
  control_label = outcome_lv[1],
  model = c("linear", "logistic", "rf"),
  subset_sizes = NULL,
  metric = NULL,
  ntree = 500,
  nodesize = NULL,
  cv_method = c("repeated_kfold", "kfold", "loocv"),
  n_fold = 5,
  n_repeat = 5,
  seed = NULL
)
```

Arguments

`data`	A data.frame (or matrix) in wide format, one row per observation. Typically the training half of a `split_data()` result.
`outcome`	The outcome, either the name of a column of `data` or a vector with one entry per row.
`predictors`	Candidate column names, or `NULL` for every column of `data` except the outcome. Numeric, logical, factor and character columns are all accepted.
`outcome_lv`	For a two-class outcome, the two classes with the reference first. `NULL` sorts them, which puts `"control"` before `"treated"` and `0` before `1`. Naming it is also what tells this function that a numeric column of zeroes and ones is two classes rather than two numbers.
`control_label`	The reference class on its own, for when the other one needs no saying. Defaults to `outcome_lv[1]`, so a call that names one of the two names the reference either way; naming both and disagreeing is an error.
`model`	What is fitted inside the search: `"linear"` for a continuous outcome, `"logistic"` for a two-class one, or `"rf"` for a random forest over either.
`subset_sizes`	Subset sizes to score, or `NULL` for a ladder that is dense where the answer usually is — every size up to ten, then 15, 20, 30, 50 and 100 — capped at the number of candidates. The full set is always scored, since keeping everything is the option a selection is being compared against.
`metric`	Which resampled number chooses the size: `"RMSE"`, `"Rsquared"` or `"MAE"` for a regression, `"Accuracy"` or `"Kappa"` for a classification, or `NULL` for the first of each. Whether it is maximised or minimised follows from the metric and is reported in `parameters`.
`ntree, nodesize`	Trees to grow and the smallest leaf to split, used by `model = "rf"` and ignored otherwise. `NULL` leaves `nodesize` at 1 for a classification and 5 for a regression, which is `randomForest()`'s own rule.
`cv_method`	Resampling scheme: `"repeated_kfold"` for `n_repeat` runs of `n_fold`-fold cross-validation, `"kfold"` for one, or `"loocv"` for leave-one-out.

n_fold	Folds per run, used by "repeated_kfold" and "kfold".
n_repeat	Number of runs, used by "repeated_kfold".
seed	Seed for the fold assignment, or NULL to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

The input is the wide format the model functions take, **one row per observation with one column as the outcome**, and is normally the training half of a split_data() result. Selecting on data a model is later scored on is how a selection flatters itself; see the details.

Value

An object of class sa_selection, a plain list.

analysis "rfe".

candidates The predictors that were offered, most important first. This is the row order ranking follows, and what terms is to a model.

design What the search saw: the outcome and its outcome_type, outcome_lv with n_events and event_rate for a classification, the row counts n_obs, n_used and n_dropped, the predictors in the order they arrived, any dropped_predictors, and predictor_lv for those that are factors.

parameters The choices as they were used: model, metric and maximize, the forest's ntree and nodesize when there was one, the resampling scheme with NA where it used none of an argument, and seed.

selected The predictors of the winning size, most important first. These are the names to hand to predictors = in a fit_*() call.

ranking One row per candidate: candidates, the estimate it was ranked by averaged over the resamples, its rank, and whether it was selected.

profile One row per subset size that was scored: n_vars, one column per metric with its standard deviation over the resamples, and chosen, which is TRUE on exactly one row.

resampling One row per resample at the chosen size.

engine What ran the search, including importance, the name of what ranking$estimate measures.

fit The caret::rfe() object. This is the slot that is not portable; dropping it leaves an object that writes out as JSON.

metadata Package version, R version, platform and timestamp.

What is resampled

The elimination is inside the resampling, not before it. Each fold ranks the predictors on its own training rows, peels them down to each size in turn, and scores every size on rows it did not rank on, so a predictor that looks useful only on the rows that chose it is caught. Ranking once on all the rows and cross-validating afterwards is the mistake this ordering exists to avoid, and it is why there

is no cv argument: an elimination with nothing held out has no score to choose a size by, so it is not a shorter version of this function but a different and wrong one.

What the resampling cannot do is make the reported score an honest estimate of the selected model. The size was chosen because it scored best, so profile reads high at the size it picked, for the same reason a maximum of noisy numbers is above their mean. Score the selection on the test half of split_data(), which the search never saw, and read profile as the shape of the search rather than as a performance claim.

The ranking is computed once per fold, at the full set of predictors, and the peeling follows it. Re-ranking after every drop is caret's rerank and is a far slower procedure — one refit per remaining predictor per fold — so it is not what this function does.

Which model does the ranking

model names what is fitted inside the search, and the outcome has to agree with it. "linear" is a continuous outcome, "logistic" a two-class one, and "rf" is either, following the outcome the way fit_rf() does. A disagreement is an error naming the model that would have fitted, rather than a silently different analysis.

What each ranks by is reported in engine$importance:

"linear", "logistic" The absolute t or Wald z statistic of each coefficient, largest first. Not the coefficient itself, which is an effect per unit of its predictor and so would rank by the units the predictors were measured in; the statistic divides that by its own standard error and has no units left. A factor is ranked as one column, by the largest statistic among its levels, and a term the fit could not estimate ranks at zero, since a predictor the others already span costs nothing to drop.

"rf" Permutation importance, the loss when a predictor's values are shuffled among the rows, measured out of bag. This is the same measure fit_rf() reports as estimate, so a forest's ranking here and its importance table there can be read together.

A forest inside the search grows at randomForest()'s own mtry for each subset — the square root of the predictor count for a classification and a third of it for a regression, which is the rule fit_rf() uses — rather than at one value throughout. A fixed mtry would exceed the predictor count at the small end of the profile, where the whole question is what a handful of predictors can do.

Which class the direction is fixed on

outcome_lv is read as it is everywhere else in this package: the first level is the reference, so outcome_lv = c("control", "case") searches for the predictors of case. control_label names that same first level on its own, for the usual case where the sort has it backwards and the other level needs no saying. The two are the same statement, so passing both and disagreeing is an error rather than a precedence rule.

For this function the direction changes nothing but the reading. A ranking is a statement about how much a predictor is worth, which is the same number whichever class is called the reference; what the levels decide is which class design counts as the events, and which way a later fit_logistic_regression() on the selected predictors will read.

What is dropped before anything runs

The same listwise deletion the model functions use: rows that are missing the outcome or any candidate go before the folds are drawn, so every fold sees the same rows, and `design$n_dropped` says how many went. A candidate that takes a single value is left out with a message, since a column with nothing in it cannot be eliminated for a reason.

Portability

Everything but `$fit` is a scalar, a character vector, a named list or a data.frame, so dropping that one slot leaves an object that writes out as JSON. In a Python transcription this is `sklearn.feature_selection.RFECV`.

See Also

`split_data()`, which defines the rows a selection should be run on, `fit_rf()` and `fit_linear_regression()` for fitting the predictors it kept, and `fit_elastic_net()`, which answers the same question from the other end by shrinking a coefficient to exactly zero rather than by dropping a column.

Examples

```
## Linear RFE on a small candidate set (fast enough for examples).
res <- perform_rfe(mtcars, outcome = "mpg",
                   predictors = c("wt", "hp", "disp", "qsec"),
                   cv_method = "kfold", n_fold = 3, seed = 1)
res$selected


res_full <- perform_rfe(mtcars, outcome = "mpg",
                        predictors = c("wt", "hp", "disp", "qsec", "drat",
                                       "carb"),
                        cv_method = "kfold", n_fold = 3, seed = 1)
fit_linear_regression(mtcars, outcome = "mpg", predictors = res_full$selected,
                      cv = FALSE)$coefficients

## Forest ranking on a two-class outcome.
iris2 <- iris[iris$Species != "setosa", ]
perform_rfe(iris2, outcome = "Species", control_label = "versicolor",
            model = "rf", ntree = 100, cv_method = "kfold", n_fold = 3,
            seed = 1)
```

`perform_stepwise` *Stepwise feature selection by information criteria*

Description

Runs a stepwise search: the model is refitted with each candidate term taken out or put in, the move that lowers AIC or BIC the most is taken, and the search stops when no single move lowers it any further. What comes back is the predictors of the model it stopped at, the path it walked to get there, and what each candidate is worth to that model, so a set of four predictors can be read against what the other six would have cost.

Usage

```
perform_stepwise(
  data,
  outcome,
  predictors = NULL,
  outcome_lv = NULL,
  control_label = outcome_lv[1],
  model = c("linear", "logistic"),
  criterion = c("AIC", "BIC"),
  direction = c("backward", "both", "forward")
)
```

Arguments

`data`	A data.frame (or matrix) in wide format, one row per observation. Typically the training half of a `split_data()` result.
`outcome`	The outcome, either the name of a column of `data` or a vector with one entry per row.
`predictors`	Candidate column names, or `NULL` for every column of `data` except the outcome. Numeric, logical, factor and character columns are all accepted; a factor is one candidate however many dummy columns it becomes.
`outcome_lv`	For a two-class outcome, the two classes with the reference first. `NULL` sorts them, which puts `"control"` before `"treated"` and `0` before `1`. Naming it is also what tells this function that a numeric column of zeroes and ones is two classes rather than two numbers.
`control_label`	The reference class on its own, for when the other one needs no saying. Defaults to `outcome_lv[1]`, so a call that names one of the two names the reference either way; naming both and disagreeing is an error.
`model`	What is fitted at every step: `"linear"` for a continuous outcome or `"logistic"` for a two-class one.
`criterion`	Which criterion the moves are judged by, `"AIC"` or `"BIC"`. The search is the same one either way; what changes is the charge per parameter, 2 against `log(n)`.
`direction`	`"backward"` starts at every candidate and only drops, `"forward"` starts at the intercept and only adds, and `"both"` starts at every candidate and may put a dropped term back.

Details

The input is the wide format the model functions take, **one row per observation with one column as the outcome**, and is normally the training half of a `split_data()` result. Selecting on data a model is later scored on is how a selection flatters itself; see the details.

Value

An object of class `sa_selection`, a plain list.

`analysis` `"stepwise"`.

`candidates` The predictors that were offered, most important first. This is the row order `ranking` follows, and what `terms` is to a model.

`design` What the search saw: the `outcome` and its `outcome_type`, `outcome_lv` with `n_events` and `event_rate` for a classification, the row counts `n_obs`, `n_used` and `n_dropped`, the `predictors` in the order they arrived, any `dropped_predictors`, and `predictor_lv` for those that are factors.

`parameters` The choices as they were used: `model`, `criterion`, `maximize`, which is `FALSE` because a smaller criterion is a better model, `k`, the charge per parameter it levied, and `direction`.

`selected` The predictors of the model the search stopped at, most important first. These are the names to hand to `predictors =` in a `fit_*()` call.

`ranking` One row per candidate: `candidates`, the `estimate` it was ranked by, its `rank`, and whether it was `selected`. The estimate is what leaving that one predictor out of the selected model costs the criterion, so it is positive for a predictor worth keeping and negative for one worth leaving out.

`profile` One row per step of the path: `n_vars`, both `AIC` and `BIC` of the model at that step, the `step` that was taken to reach it, and `chosen`, which is `TRUE` on the last row.

`resampling` `NULL`. Nothing was resampled.

`engine` What ran the search, including `importance`, the name of what `ranking$estimate` measures.

`fit` The `stats::step()` result, which is the selected `lm` or `glm` with the path attached as `$anova`. This is the slot that is not portable; dropping it leaves an object that writes out as JSON.

`metadata` Package version, R version, platform and timestamp.

What the criterion charges

Both criteria are the fit's own log likelihood with a charge levied per parameter, and they differ only in the size of the charge: `"AIC"` levies 2 and `"BIC"` levies `log(n)`, so past seven observations BIC charges more and keeps fewer predictors. Smaller is better for both, which is why `maximize` is `FALSE` and is reported rather than asked for. The charge as it was used is `parameters$k`.

Whichever one searches, `profile` reports both at every step, so a path chosen by AIC can be read against what BIC would have said about the same models. They are `stats::AIC()`'s and `stats::BIC()`'s own numbers, the ones `fit_linear_regression()` reports in `fit_stats`, so a step of the path and a fitted model are the same number. `stats::step()` itself searches on `stats::extractAIC()`'s scale, which for a linear model differs from `stats::AIC()`'s by a constant that is the same for every model on `n` rows; the path is therefore ordered identically and only the printed values differ.

A criterion is comparable across models only when they were fitted to the same rows, which is what the listwise deletion below is for. It is not comparable across outcomes, across transformations of an outcome, or between a linear and a logistic fit, so the numbers in `profile` say which of these models to prefer and nothing more.

What the search is not

The criterion is computed on the rows the model was fitted to, and the model was kept because it scored best on them. Three things follow, and none of them is a fault of the implementation:

The criterion is not a validation Score the selection on the test half of `split_data()`, which the search never saw. That is the only number here that is a claim about new rows, and it is why this function has no `cv` argument: cross-validating a penalised likelihood computed on the training rows would resample the wrong quantity.

The p-values of the selected model are no longer honest A coefficient kept because it was significant enough to survive is being tested against a null it was already screened on, so the p-value `fit_linear_regression()` reports for it on the same rows reads smaller than it is. Refit on held-out rows, or read the fit as a description of these rows rather than as a test.

The path is greedy One move is taken at a time, so a pair of predictors that is worth keeping only together can be dropped one at a time and never come back. `direction = "both"` reconsiders a dropped term at every later step, which is what it is for; nothing short of fitting every subset removes the problem entirely.

`fit_elastic_net()` answers the same question without a path at all, by shrinking a coefficient to exactly zero, and `perform_rfe()` answers it with a resampled score rather than with a penalty.

Where the search starts and which way it moves

`"backward"` starts at every candidate and only drops. `"forward"` starts at the intercept and only adds. `"both"` starts at every candidate and may add a term back after dropping it, so its path can visit the same size twice; that is the one direction whose `profile` is not a ladder.

Every direction is bounded by the same two models: the intercept alone below and all of `predictors` above. A search that walks back to the intercept has kept nothing, which is an answer — no candidate pays for itself at this charge — but not one this contract can carry, since `selected` would be empty. It is an error saying so rather than a result with a hole in it.

Which model does the searching

`model` names what is fitted at every step, and the outcome has to agree with it: `"linear"` is a continuous outcome and `"logistic"` a two-class one. A disagreement is an error naming the model that would have fitted, rather than a silently different analysis. There is no `"rf"` here, unlike `perform_rfe()`: a criterion is a likelihood with a charge against its parameter count, and a forest has neither.

`outcome_lv` is read as it is everywhere else in this package: the first level is the reference, so `outcome_lv = c("control", "case")` searches for the predictors of `case`, and `control_label` names that first level on its own. For the search itself the direction changes nothing, since the likelihood of a model is the same number whichever class is called the reference; what the levels decide is which class a later `fit_logistic_regression()` on `$selected` reads.

What is dropped before anything runs

The same listwise deletion the model functions use: rows missing the outcome or any candidate go first, and `design$n_dropped` says how many went. Here it is also what makes the search legible, since a criterion compares models only when they were fitted to the same rows, and leaving the deletion to the engine would have each step fitted to whatever its own terms happened to be complete on. A candidate that takes a single value is left out with a message, since a column with nothing in it cannot earn a parameter.

Portability

Everything but `$fit` is a scalar, a character vector, a named list or a data.frame, so dropping that one slot leaves an object that writes out as JSON.

See Also

`split_data()`, which defines the rows a selection should be run on, `perform_rfe()`, which chooses by a resampled score rather than by a penalty, `fit_elastic_net()`, which chooses by shrinking a coefficient to exactly zero, and `fit_linear_regression()` and `fit_logistic_regression()` for fitting the predictors it kept.

Examples

```
## Six candidates and one continuous outcome. The path says what was dropped and
## when, and `$selected` is where the search stopped.
res <- perform_stepwise(mtcars, outcome = "mpg",
                    predictors = c("wt", "hp", "disp", "qsec", "drat", "carb"))
res
res$selected
res$profile

## The same search at a heavier charge per parameter keeps fewer predictors.
perform_stepwise(mtcars, outcome = "mpg",
             predictors = c("wt", "hp", "disp", "qsec", "drat", "carb"),
             criterion = "BIC")$selected

## The selection is a set of column names, so it goes straight back into a fit.
fit_linear_regression(mtcars, outcome = "mpg", predictors = res$selected,
                      cv = FALSE)$coefficients

## A two-class outcome, with the reference named on its own.
iris2 <- iris[iris$Species != "setosa", ]
perform_stepwise(iris2, outcome = "Species", control_label = "versicolor",
             model = "logistic")
```

`perform_tsne` *Embed samples or features with t-SNE*

Description

Places each point in two or three dimensions so that the points it was near in the full feature space stay near it, by t-distributed stochastic neighbour embedding. What comes back is a picture and the coordinates to draw it with: an axis is not a direction the way a principal component is, so there is nothing to read off one but position.

Usage

```
perform_tsne(
  data,
  feats = NULL,
  embedding_scale = c("samples", "features"),
  center = TRUE,
  scale = TRUE,
  n_dim = 2,
  perplexity = NULL,
  theta = 0.5,
  seed = NULL
)
```

Arguments

`data`	A data.frame or a matrix in wide format, one row per sample and one column per feature. Row names are kept as the sample labels, repeated ones included; rows without a name are labelled by position.
`feats`	Column names to embed, or `NULL` for every numeric column of `data`. A non-numeric column is left out with a message.
`embedding_scale`	Which margin becomes the points of the picture: `"samples"`, the default, for one point per row of `data`, or `"features"` for one point per column, in which case the standardised matrix is transposed before it is embedded. `design$point_type` reports which it was.
`center`, `scale`	Whether to centre each feature and divide it by its standard deviation before embedding. Both always apply to the **columns of** `data`, whatever `embedding_scale` is.
`n_dim`	How many dimensions to embed into. `Rtsne` embeds into at most 3.
`perplexity`	The neighbourhood size, or `NULL` to read one off the number of points. It is roughly how many neighbours each point is asked to keep close, and `Rtsne` requires `3 * perplexity <= n - 1`.
`theta`	Barnes-Hut approximation. `0` is the exact gradient and slow, and larger values trade accuracy for speed.

`seed` Seed for the embedding, or `NULL` to use the stream as it stands. Supplying a seed does not disturb the caller: the previous random number state is put back when the function returns. Without one, two runs on the same data give two different pictures, which is a property of the method rather than a fault.

Details

The input is the wide format the comparison functions take: **one row per sample and one column per feature**, whichever margin is being embedded. What comes back has one row per point, in one order every table follows, which is what makes an embedding plottable against anything else read from the same frame.

Value

An object of class `sa_reduction`, a plain list.

`analysis` `"tsne"`.

`points` Labels of the things that became points — samples or features, as `embedding_scale` asked — in the row order `scores` follows.

`design` What was embedded: `point_type`, either `"sample"` or `"feature"`, the `feats` kept and any `dropped_feats`, and the counts `n_samples`, `n_used`, `n_dropped` and `n_feats`. The counts describe the input rather than the picture.

`parameters` The choices as they were used rather than as they were passed, so a derived `perplexity` is the value that was derived.

`scores` Coordinates: `points` beside `tSNE1`, `tSNE2` and so on.

`engine` What computed the embedding, and which of its defaults were overridden.

`fit` The `Rtsne` object. This is the slot that is not portable; dropping it leaves an object that writes out as JSON.

`metadata` Package version, R version, platform and timestamp.

Scaling, and why it is on

Every distance this method measures is a distance across all of `feats` at once, so a feature measured in thousands would decide who is whose neighbour. `center` and `scale` are therefore on by default, as they are for `perform_pca()`, and the two functions then see literally the same matrix — which is what lets the two pictures be attributed to the methods rather than to the preprocessing.

`perform_umap()` defaults the other way, since UMAP is more often run on coordinates that already mean something, such as the components of a PCA.

Choosing the margin

`embedding_scale = "features"` embeds the features instead, and unlike `perform_pca()` this really does transpose: t-SNE embeds the rows it is handed and has no second answer to read off the same fit. The features are standardised first and the transpose is then embedded as it stands, which is what makes this the same margin `perform_pca(embedding_scale = "features")` reports on. Standardising after the transpose would standardise samples, which is what `perform_tsne(t(data))` does and why it is not the same call.

A feature margin needs enough features to be worth drawing. `perplexity` is read off the number of points, so 8 features force a perplexity of 2 and a message says so; at that size the loadings of a PCA are the whole answer and an embedding has nothing to add. Sixty features derive a perplexity of 19, and on a simulation with three planted correlation blocks the embedding then recovers them exactly.

What is dropped before anything runs

`Rtsne` does not accept a missing value, so rows that are not complete and finite across `feats` are dropped before it is called, and `design$n_dropped` reports how many went. A feature that takes a single value cannot be scaled, so with `scale = TRUE` it is left out with a message and named in `design$dropped_feats`.

Portability

Everything but `$fit` is a data.frame, a character vector or a named list, so dropping that one slot leaves an object that writes out as JSON. In a Python transcription this is `sklearn.manifold.TSNE`.

See Also

`perform_pca()`, which answers about the same points in a way that can say which features moved them, and `perform_umap()`, the other embedding.

Examples

```
## The group was never shown to the embedding, so the split is its to find.
sim <- simulate_two_groups(n_feats = 30, n_up = 5, n_down = 5, seed = 3)
res <- perform_tsne(sim$args$data, seed = 1)
head(res$scores)


plot(res$scores[c("tSNE1", "tSNE2")],
     col = as.integer(factor(sim$args$group)), pch = 16)
pca <- perform_pca(sim$args$data)
table(group = sim$args$group, side = pca$scores$PC1 > 0)

## Feature-scale embedding over planted correlation blocks.
cor_mat <- make_block_cor(
  n_features = 60,
  blocks = list(list(features = 1:20, cor = 0.8),
                list(features = 21:40, cor = 0.6),
                list(features = 41:60, cor = 0.4))
)
blocks <- simulate_classification(n_pred = 60, cor_mat = cor_mat,
                                  seed = 2026)$args$data
by_feat <- perform_tsne(blocks, feats = paste0("x_", 1:60),
                        embedding_scale = "features", seed = 1)
plot(by_feat$scores[c("tSNE1", "tSNE2")],
     col = rep(1:3, each = 20), pch = 16)
```

perform_umap *Embed samples or features with UMAP*

Description

Places each point in a few dimensions so that the neighbourhood structure of the full feature space survives, by uniform manifold approximation and projection. What comes back is a picture and the coordinates to draw it with: an axis is not a direction the way a principal component is, so there is nothing to read off one but position.

Usage

```
perform_umap(
  data,
  feats = NULL,
  embedding_scale = c("samples", "features"),
  center = FALSE,
  scale = FALSE,
  n_dim = 2,
  n_neighbors = NULL,
  min_dist = 0.1,
  method = c("naive", "umap-learn"),
  metric = c("euclidean", "manhattan", "cosine", "pearson"),
  seed = NULL
)
```

Arguments

data	A data.frame or a matrix in wide format, one row per sample and one column per feature. Row names are kept as the sample labels, repeated ones included; rows without a name are labelled by position.
feats	Column names to embed, or NULL for every numeric column of data. A non-numeric column is left out with a message.
embedding_scale	Which margin becomes the points of the picture: "samples", the default, for one point per row of data, or "features" for one point per column, in which case the matrix is transposed before it is embedded. design$point_type reports which it was.
center, scale	Whether to centre each feature and divide it by its standard deviation before embedding. Both are off by default, unlike perform_pca() and perform_tsne(); see the details. They always apply to the **columns of** data, whatever embedding_scale is.
n_dim	How many dimensions to embed into.
n_neighbors	The neighbourhood size, or NULL to read one off the number of points. Small values follow local detail and large ones the overall shape.

`min_dist` How tightly points that belong together are allowed to be packed.

`method` Which `umap::umap()` backend to use: `"naive"`, the default, is the package's pure R implementation and needs no Python; `"umap-learn"` calls the reference implementation through `reticulate` and requires `umap-learn` installed in a Python environment the package can see.

`metric` Distance neighbours are measured with. `"cosine"` and `"pearson"` compare the shape of a row rather than its size, which is why the two scaling flags are off by default.

`seed` Seed for the embedding, or `NULL` to use the stream as it stands. Supplying a seed does not disturb the caller: the previous random number state is put back when the function returns. It buys less here than it does in `perform_tsne()`, since the `umap` package restores the stream itself, so two seedless runs from the same state already agree.

Details

The input is the wide format the comparison functions take: **one row per sample and one column per feature**, whichever margin is being embedded. What comes back has one row per point, in one order every table follows, which is what makes an embedding plottable against anything else read from the same frame.

Value

An object of class `sa_reduction`, a plain list.

`analysis` `"umap"`.

`points` Labels of the things that became points — samples or features, as `embedding_scale` asked — in the row order `scores` follows.

`design` What was embedded: `point_type`, either `"sample"` or `"feature"`, the `feats` kept and any `dropped_feats`, and the counts `n_samples`, `n_used`, `n_dropped` and `n_feats`. The counts describe the input rather than the picture.

`parameters` The choices as they were used rather than as they were passed, so a derived `n_neighbors` is the value that was derived.

`scores` Coordinates: `points` beside `UMAP1`, `UMAP2` and so on.

`engine` What computed the embedding, and which of its defaults were overridden.

`fit` The `umap` object. This is the slot that is not portable; dropping it leaves an object that writes out as JSON.

`metadata` Package version, R version, platform and timestamp.

Scaling, and why it is off

`center` and `scale` are `FALSE` here and `TRUE` in `perform_pca()` and `perform_tsne()`. The difference is `metric`: `"cosine"` and `"pearson"` compare the shape of a row rather than its size, so they have already answered the question standardising would answer, and the `umap` package does not standardise either. The default is therefore the engine's own picture of the data as it arrived.

What that leaves the caller is one decision rather than none. With `metric = "euclidean"` or `"manhattan"` on features that are not measured on a common scale, the feature with the widest units decides who is whose neighbour, and `scale = TRUE` is what the picture needs — it is also what makes this embedding comparable with a `perform_pca()` or `perform_tsne()` of the same data, since those two standardise by default.

One consequence of the default is that a feature of no variance is kept. It can be kept because nothing divides by its standard deviation; it contributes nothing to any distance either way. With `scale = TRUE` it is left out with a message and named in `design$dropped_feats`.

Choosing the margin

`embedding_scale = "features"` embeds the features instead, and unlike `perform_pca()` this really does transpose: UMAP embeds the rows it is handed and has no second answer to read off the same fit. `center` and `scale`, if they are turned on, still apply to the **features** and the transpose is then embedded as it stands, which is what makes this the same margin `perform_pca(embedding_scale = "features")` reports on. `perform_umap(t(data))` with `scale = TRUE` would standardise samples instead, which is a third analysis.

A feature margin needs enough features to be worth drawing. `n_neighbors` is read off the number of points, so 8 features force a neighbourhood of 8 and a message says so; at that size the loadings of a PCA are the whole answer.

What is dropped before anything runs

`umap` does not accept a missing value, so rows that are not complete and finite across `feats` are dropped before it is called, and `design$n_dropped` reports how many went. This is the listwise deletion the rest of the package uses; nothing is imputed.

Portability

Everything but `$fit` is a data.frame, a character vector or a named list, so dropping that one slot leaves an object that writes out as JSON. The `umap` object is the one genuinely unportable thing this package produces: its `config$metric.function` is a function. In a Python transcription this is the separate `umap-learn` package rather than anything in scikit-learn.

See Also

`perform_pca()`, which answers about the same points in a way that can say which features moved them, and `perform_tsne()`, the other embedding.

Examples

```
## The group was never shown to the embedding, so the split is its to find.
sim <- simulate_two_groups(n_feats = 30, n_up = 5, n_down = 5, seed = 3)
res <- perform_umap(sim$args$data, seed = 1)
head(res$scores)


plot(res$scores[c("UMAP1", "UMAP2")],
     col = as.integer(factor(sim$args$group)), pch = 16)
scaled <- perform_umap(sim$args$data, scale = TRUE, seed = 1)
```

```
## Feature-scale embedding over planted correlation blocks.
cor_mat <- make_block_cor(
  n_features = 60,
  blocks = list(list(features = 1:20, cor = 0.8),
                list(features = 21:40, cor = 0.6),
                list(features = 41:60, cor = 0.4))
)
blocks <- simulate_classification(n_pred = 60, cor_mat = cor_mat,
                                  seed = 2026)$args$data
by_feat <- perform_umap(blocks, feats = paste0("x_", 1:60),
                        embedding_scale = "features", scale = TRUE, seed = 1)
plot(by_feat$scores[c("UMAP1", "UMAP2")],
     col = rep(1:3, each = 20), pch = 16)
```

`plot.sa_performance` *Draw an evaluation result*

Description

Dispatches on what was evaluated, since the two scenarios have different pictures rather than one picture with a switch: a regression is drawn against the outcome it predicted and a classification against the two classes it ranked. Call `draw_prediction_plot()` or `draw_roc_curve()` directly to reach their own arguments by name.

Usage

```
## S3 method for class 'sa_performance'
plot(x, ...)
```

Arguments

`x`	An evaluation, as returned by `evaluate_regression_models()` or `evaluate_classification_models`
`...`	Passed to whichever of the two is called.

Value

Whatever the function called returns, invisibly.

See Also

`draw_prediction_plot()` and `draw_roc_curve()`.

Examples

```
train <- mtcars[1:24, ]
test <- mtcars[25:32, ]
fit <- fit_linear_regression(train, outcome = "mpg",
                             predictors = c("wt", "hp"), cv = FALSE)
plot(evaluate_regression_models(fit, newdata = test))
```

`predict.sa_model` *Predict from a fitted model on rows it was not fitted to*

Description

Takes the data frame the fit took — the test half of a `split_data()` result, say — and answers one prediction per row of it. The columns are read by name and coded the way the fit coded them, so the rows to predict can be handed over exactly as they came, outcome column and all.

Usage

```
## S3 method for class 'sa_model'
predict(object, newdata = NULL, type = c("raw", "response", "prob"), ...)
```

Arguments

`object`	A fitted model, as returned by `fit_linear_regression()`, `fit_logistic_regression()`, `fit_elastic_net()`, `fit_rf()` or `fit_svm()`.
`newdata`	Rows to predict, a data.frame or matrix carrying the predictor columns, or `NULL` for the rows the model was fitted on.
`type`	`"raw"` for the prediction itself, a fitted value for a regression and a class label for a classification; `"response"` for the prediction on the scale of the outcome, which for a classification is the probability of `design$outcome_lv[2]`, the class the coefficients describe; or `"prob"` for one column per class.
`...`	Passed on to `caret::predict.train()`.

Details

This is the method to use rather than `predict(object$fit, newdata = )`. The engine object knows the names of the columns it was given and nothing about where they came from, which is enough for `fit_linear_regression()`, `fit_logistic_regression()` and `fit_rf()`, whose engine was handed the predictor frame itself, and not enough for `fit_elastic_net()` or `fit_svm()`, whose engines were handed a design matrix. `glmnet` and `kernlab` read that matrix by position, and `caret` prepares `newdata` for it by keeping the columns whose names it recognises, in the order `newdata` happens to hold them. A factor predictor is therefore dropped whole, since `x_cat` is not one of the names — the model has `x_catmid` and `x_cathigh` — and a set of numeric predictors in another order is silently matched to the wrong coefficients, or measured along the wrong axis of the kernel. Here the terms are rebuilt from the levels the fit recorded and put in the model's own order by name, which is what makes one call work for every model in the family.

Columns the model never saw are ignored, so the outcome column and anything else the frame carries can stay. A predictor that is absent is an error naming it, and so is a factor level the fit never saw, since neither has a coefficient to be predicted with.

There is one prediction per row of `newdata` whatever the row holds, and a row with a missing value among the predictors gets `NA`. That is the rule the fit already follows in reverse: those are the rows `design$n_dropped` counted, and they cannot be predicted for the same reason they could not be fitted. Saying so with an `NA` in place keeps the answer aligned with `newdata`, which a shorter vector would not be. It also has to be said explicitly, because a penalized fit would otherwise predict some of them: a coefficient of exactly zero drops out of the sparse product before the missing value it multiplies is ever read, so whether a row could be predicted would depend on which predictor the hole fell in.

Value

One prediction per row of `newdata`: an unnamed numeric vector for a regression and for `type = "response"`, a factor at the levels of `design$outcome_lv` for `type = "raw"` on a classification, and a data.frame of one column per class for `type = "prob"`. Rows that are not complete across the predictors are `NA` throughout.

See Also

`fit_elastic_net()`, `fit_rf()`, `fit_svm()`, `split_data()`, which draws the rows to predict, and `predict.sa_fit()` for the engine object in `$fit`, which is `caret`'s own method and takes what `caret` prepared.

Examples

```
sp <- split_data(mtcars, seed = 1)
train <- sp$datasets[[1]]$train_data
test <- sp$datasets[[1]]$test_data

fit <- fit_linear_regression(train, outcome = "mpg",
                             predictors = c("wt", "hp"), cv = FALSE)
sqrt(mean((test$mpg - predict(fit, newdata = test))^2))

## A factor predictor is what `predict(fit$fit, )` cannot be given on a
## penalized fit: the model has one term per level, and the frame has the
## column the levels came from.
train$cyl <- factor(train$cyl)
test$cyl <- factor(test$cyl, levels = levels(train$cyl))
pen <- fit_elastic_net(train, outcome = "mpg",
                       predictors = c("wt", "hp", "cyl"),
                       penalty = "lasso", lambda = 0.5, cv = FALSE)
predict(pen, newdata = test)

## On a classification, `type = "response"` is the probability of the second
## level of `outcome_lv`, the one the coefficients describe.
iris2 <- iris[iris$Species != "setosa", ]
clf <- fit_logistic_regression(iris2, outcome = "Species",
                               predictors = "Petal.Length",
                               outcome_lv = c("versicolor", "virginica"),
```

```
                                cv = FALSE)
head(predict(clf, newdata = iris2, type = "response"))
```

`print.sa_categorical` *Print a categorical comparison*

Description

Summarises the table, the null hypothesis it was held against, the tests run on it and the association measures, rather than printing the cell table. `as.table()` gives the table as counts, `$cells` the residuals and `$association` the measures with their intervals.

Usage

```
## S3 method for class 'sa_categorical'
print(x, alpha = 0.05, ...)
```

Arguments

`x`	A categorical comparison, as returned by `compare_categorical_groups()`.
`alpha`	Threshold the reported verdict on each test is read at.
`...`	Ignored, present for consistency with `print()`.

Value

`x` invisibly.

Examples

```
compare_categorical_groups(
  data.frame(smoker = c(rep("y", 30), rep("n", 30)),
             grade  = c(rep(c("high", "low"), c(22, 8)),
                        rep(c("high", "low"), c(9, 21))))
)
```

print.sa_categorical_significance *Print a categorical significance verdict*

Description

Summarises the rule that was applied and what cleared it, rather than printing the table itself. Reach into `x$significance` for that.

Usage

```
## S3 method for class 'sa_categorical_significance'
print(x, ...)
```

Arguments

`x`	An `sa_categorical_significance` object, as returned by `estimate_categorical_significance()`.
`...`	Ignored, present for consistency with `print()`.

Value

`x` invisibly.

Examples

```
res <- compare_categorical_groups(
  data.frame(smoker = rep(c("y", "n"), each = 30),
             grade  = c(rep(c("high", "low"), c(22, 8)),
                        rep(c("high", "low"), c(9, 21))))
)
estimate_categorical_significance(res)
estimate_categorical_significance(res, by = "table")
```

print.sa_cluster *Print a clustering*

Description

Summarises what was clustered and what came of it, rather than printing the label of every point. Those are in `x$assignments`, and the engine object is `x$fit`.

Usage

```
## S3 method for class 'sa_cluster'
print(x, n = 10L, ...)
```

Arguments

x	A clustering, as returned by cluster_hclust(), cluster_kmeans(), cluster_dbscan() or cluster_snn().
n	Maximum number of clusters to report the size of. The rest are counted.
...	Ignored, present for consistency with print().

Value

x invisibly.

Examples

```
cluster_kmeans(iris[1:4], n_clust = 3, seed = 1)
```

---

print.sa_comparison *Print a comparison result*

Description

Summarises which tests were run and how many features each one called significant, rather than printing the tables themselves. Reach into x$tests for those, and into x$posthoc for the pairwise stage.

Usage

```
## S3 method for class 'sa_comparison'
print(x, alpha = 0.05, ...)
```

Arguments

x	A comparison result, as returned by compare_two_groups(), compare_multiple_groups() or compare_one_sample().
alpha	Threshold applied to pval_adj for the significant counts.
...	Ignored, present for consistency with print().

Value

x invisibly.

Examples

```
iris2 <- iris[iris$Species != "setosa", ]
compare_two_groups(iris2, c("Sepal.Length", "Petal.Length"),
                   iris2$Species, c("versicolor", "virginica"))
```

print.sa_diagnosis *Print a distribution diagnosis*

Description

Reports how many features failed each check rather than printing the tables. Reach into `$normality`, `$variance` and `$outliers` for those.

Usage

```
## S3 method for class 'sa_diagnosis'
print(x, ...)
```

Arguments

`x`	A diagnosis, as returned by `diagnose_distribution()`.
`...`	Ignored, present for consistency with `print()`.

Value

`x` invisibly.

Examples

```
diagnose_distribution(iris, c("Sepal.Length", "Petal.Length"), iris$Species)
```

print.sa_model *Print a fitted model*

Description

Summarises what was fitted to what, and how it did, rather than printing every table. The coefficient table is in `x$coefficients`, which is also what `coef()` answers with; the resampled folds are in `x$resampling`, and the engine object `predict()` and `summary()` take is `x$fit`.

Usage

```
## S3 method for class 'sa_model'
print(x, n = 10L, ...)
```

Arguments

`x`	A fitted model, as returned by `fit_linear_regression()`, `fit_logistic_regression()`, `fit_elastic_net()`, `fit_rf()` or `fit_svm()`.
`n`	Maximum number of coefficient rows to show. The rest are counted.
`...`	Ignored, present for consistency with `print()`.

Value

x invisibly.

Examples

```
fit_linear_regression(mtcars, outcome = "mpg",
                      predictors = c("wt", "hp"), cv = FALSE)
```

---

print.sa_performance *Print an evaluation result*

Description

Summarises what was scored on what, and how each model did, rather than printing every table. The per-model scores are in `x$metrics`, what each model did against the baseline is in `x$comparisons`, and the predictions themselves are in `x$predictions`.

Usage

```
## S3 method for class 'sa_performance'
print(x, n = 10L, ...)
```

Arguments

x	An evaluation, as returned by `evaluate_regression_models()` or `evaluate_classification_models`
n	Maximum number of models to show. The rest are counted.
...	Ignored, present for consistency with `print()`.

Value

x invisibly.

Examples

```
## Two models of the same outcome, scored on rows neither was fitted on.
train <- mtcars[1:24, ]
test <- mtcars[25:32, ]
full <- fit_linear_regression(train, outcome = "mpg",
                              predictors = c("wt", "hp", "disp"),
                              cv = FALSE)
small <- fit_linear_regression(train, outcome = "mpg",
                               predictors = "wt", cv = FALSE)
evaluate_regression_models(full, list(weight_only = small), newdata = test)
```

print.sa_reduction *Print a dimensionality reduction*

Description

Summarises what was reduced and how far, rather than printing the coordinates. Those are in `x$scores`, and the engine object is `x$fit`.

Usage

```
## S3 method for class 'sa_reduction'
print(x, n = 3L, ...)
```

Arguments

`x`	A reduction, as returned by `perform_pca()`, `perform_tsne()` or `perform_umap()`.
`n`	Maximum number of components to report the variance of. The rest are counted. Ignored by an embedding, which has no components.
`...`	Ignored, present for consistency with `print()`.

Value

`x` invisibly.

Examples

```
perform_pca(iris[1:4])
```

print.sa_selection *Print a feature selection*

Description

Summarises what was searched and what was kept, rather than printing every table. The score of every model the search compared is in `x$profile`, which is what says whether the answer won by much or by nothing, and the engine object is `x$fit`.

Usage

```
## S3 method for class 'sa_selection'
print(x, n = 10L, ...)
```

Arguments

x	A selection, as returned by perform_rfe() or perform_stepwise().
n	Maximum number of candidates to show. The rest are counted.
...	Ignored, present for consistency with print().

Value

x invisibly.

Examples

```
perform_rfe(mtcars, outcome = "mpg", predictors = c("wt", "hp", "disp"),
            cv_method = "kfold", n_fold = 3, seed = 1)

## The same method, on a search that walked a path rather than a ladder of
## subset sizes and that held no rows out to score it.
perform_stepwise(mtcars, outcome = "mpg", predictors = c("wt", "hp", "disp"))
```

---

print.sa_significance *Print a significance verdict*

Description

Summarises the rule that was applied and how many features cleared it, rather than printing the table itself. Reach into x$significance for that.

Usage

```
## S3 method for class 'sa_significance'
print(x, ...)
```

Arguments

x	An sa_significance object, as returned by estimate_significance().
...	Ignored, present for consistency with print().

Value

x invisibly.

Examples

```
iris2 <- iris[iris$Species != "setosa", ]
res <- compare_two_groups(iris2, c("Sepal.Length", "Petal.Length"),
                          iris2$Species, c("virginica", "versicolor"))
estimate_significance(res, log2fc_cutoff = 0.1)
```

print.sa_split *Print a train/test split*

Description

Summarises what the split was made on and how large each half came out, rather than printing the data. The frames themselves are in `x$datasets[[i]]$train_data` and `$test_data`.

Usage

```
## S3 method for class 'sa_split'
print(x, ...)
```

Arguments

`x`	A split, as returned by `split_data()`.
`...`	Ignored, present for consistency with `print()`.

Value

`x` invisibly.

Examples

```
split_data(iris, stratified = "Species", times = 2, seed = 1)
```

screen_outliers *Flag candidate outliers without removing them*

Description

Screens each feature, or each feature within each group level, and returns one row per flagged observation. Nothing is deleted and no analysis changes as a result. Which observations belong in a data set is a decision about the experiment rather than about the arithmetic, and the package does not make it on the user's behalf.

Usage

```
screen_outliers(
  data,
  feats,
  group = NULL,
  group_lv = NULL,
  criterion = c("iqr", "robust_z", "grubbs"),
  iqr_multiplier = 1.5,
```

```
  z_threshold = 3.5,
  alpha = 0.05
)
```

Arguments

data	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
feats	Character vector of numeric column names in data to screen.
group	Optional grouping vector with one entry per row of data. When supplied, each feature is screened within each level separately, which is what keeps a genuine group difference from being read as a set of outliers.
group_lv	Group levels to keep, in display order. Defaults to the sorted unique values of group. Rows belonging to any other level are dropped.
criterion	"iqr", "robust_z" or "grubbs".
iqr_multiplier	Fence width for criterion = "iqr".
z_threshold	Cut-off for criterion = "robust_z". The 3.5 default is the Iglewicz and Hoaglin recommendation.
alpha	Significance level for criterion = "grubbs".

Details

Three rules are available and they do not agree with each other, which is the point of naming the one used:

"iqr" Anything past Q1 - k * IQR or Q3 + k * IQR. The rule behind the whiskers of a boxplot. It makes no distributional assumption and flags a fixed share of any long-tailed sample.

"robust_z" Distance from the median in units of the median absolute deviation. Uses the median and MAD rather than the mean and standard deviation because one extreme value inflates the standard deviation enough to hide itself.

"grubbs" Tests only the single most extreme observation, against the null that the sample is normal. It is the only rule that produces a p-value and the only one that assumes a distribution.

Value

A data.frame with one row per flagged observation and the columns features, group (NA when no group was given), row (the row number in the data that was passed in), value and score. The score is the quantity the rule thresholded: distance past the nearer quartile in IQR units for "iqr", the robust z for "robust_z", the Grubbs statistic for "grubbs". Zero rows means nothing was flagged. The criterion and its thresholds are attached as attributes.

References

Iglewicz, B. and Hoaglin, D. C. (1993). *How to Detect and Handle Outliers*. ASQC Quality Press.

Grubbs, F. E. (1969). Procedures for detecting outlying observations in samples. *Technometrics*, 11(1), 1-21.

See Also

diagnose_distribution(), which runs this alongside the normality and variance checks, and summarize_descriptive_stats(), whose out_lower_bound and out_upper_bound columns are the same IQR fences.

Examples

```
feats <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")

## Across the whole data set, sepal width is the only measurement with
## observations past the fences.
screen_outliers(iris, feats)

## Within species, petal measurements pick up flags that the pooled screen
## hid: the between-species spread was widening the fences.
screen_outliers(iris, feats, iris$Species)

## A stricter rule and a distributional one disagree, which is why the
## criterion is named rather than assumed.
nrow(screen_outliers(iris, feats, criterion = "robust_z"))
screen_outliers(iris, feats, criterion = "grubbs")
```

---

simulate_categorical_groups

Simulate a contingency table whose association is known

Description

Generates two or more categorical variables with a chosen amount of association planted between them, and returns the planted answer alongside the data. Every quantity compare_categorical_groups() estimates can then be checked against what was actually put there, which is what a real data set can never offer.

Usage

```
simulate_categorical_groups(
  n_samples = 200,
  category_lv = NULL,
  margins = NULL,
  assoc = 0.3,
  pattern = c("corner", "single", "gradient"),
  paired = FALSE,
  discordance = c(0.25, 0.1),
  seed = NULL
)
```

Arguments

n_samples	Number of observations, meaning rows for a cross-classified design and subjects for a matched one.
category_lv	Named list giving the levels of each variable, passed straight through to compare_categorical_groups() so its names are the column names of the generated data and its order fixes the reference level of each variable. A matched design needs one binary level set shared by every condition, and the default changes to reflect that when paired = TRUE.
margins	Named list of the marginal probabilities of each variable, in the order its levels are given, or NULL for a uniform margin. Each entry is normalised to sum to one, so relative weights are enough.
assoc	How much association to plant, from 0 for exact independence to 1 for the largest margin-preserving departure the margins allow.
pattern	Which cells the association moves mass between. See "The shape of the association".
paired	Logical. If TRUE, the columns are repeated binary measurements on the same subject rather than different variables.
discordance	The two transition probabilities of a matched design, from the first level to the second and back. Their ratio is the planted paired odds ratio, and their being equal is the strict null.
seed	Seed for the draw, restored on exit, so the caller's random stream is left as it was found.

Details

The point of the exercise is the gap. A table drawn from a distribution is not that distribution: the observed Cramer's V carries a sampling error of its own, and it is biased upwards, because every departure from independence counts towards it whether it was planted or drawn. So a table simulated at assoc = 0 does not come back with an estimated association of zero, and reading that number as the answer rather than the p-value beside it is the mistake the simulator exists to make visible.

Value

A plain list, the shape every simulator in the package returns:

args data, category_lv and paired, named after the arguments of compare_categorical_groups() so that do.call(compare_categorical_groups, sim$args) runs the analysis this data was made for.

truth One row summarising what was planted: n_samples, pattern, and the population value of every association measure the comparison reports and this design defines. A cross-classification adds assoc and cramers_v, plus phi_coefficient and odds_ratio on a 2 x 2 table. A matched design adds the transition probabilities, and then either the three paired measures or the sequence of response rates. These are the values the estimates should approach as n_samples grows, and they carry no sampling error.

`truth_cell` One row per cell: `row_level`, `col_level`, `p_independent`, `p_planted`, `lift` and `expected_n`. It merges with `$cells` of the comparison on `c("row_level", "col_level")` with neither side renamed, so a residual can be read against the departure that produced it. A matched pair of conditions adds `p_symmetric` and `expected_symmetry_n`, which are what `$cells$expected` holds there: symmetry is the null that design is tested against, so it is the one the planted table has to be scored on. For three or more conditions the cells are condition by response, and there marginal homogeneity and independence are the same arithmetic, so `p_independent` already is the null.

How the association is planted

The margins are fixed first, and then mass is moved between the cells in a way that leaves them exactly where they were. That is what `assoc` scales: a perturbation matrix whose every row and column sums to zero, added to the independent joint distribution.

Keeping the margins fixed is what makes the null hypothesis the only thing that moves. A test of independence compares the table to the product of its own observed margins, so a construction that shifted the margins too would plant an association and change what it is being measured against at the same time.

`assoc` is the fraction of the largest step of that shape the table can take before a cell reaches zero, so it runs from 0 to 1 whatever the margins are. `assoc = 0` is the product of the margins **exactly**, so it is null in the strict sense and is what a type I error rate can be measured on. `assoc = 1` puts a structural zero in the table, which is the one setting where the exact test and the approximation part company sharply.

The shape of the association

`pattern` picks which cells the mass moves between. All three keep the margins fixed, so all three are the same kind of departure at different places.

`"corner"` Mass moves into the two cells where both variables sit at their first level or both at their second, and out of the two where they disagree. On a 2 x 2 table this is the whole of the association, and the odds ratio it plants is above 1.

`"single"` One level of each variable is special and the rest are alike: mass moves into the cell where both first levels meet and is taken evenly from the others. This is the shape a test of homogeneity is usually looking for, one category behaving differently from the pack.

`"gradient"` A monotone association, planted by a centred linear ramp over the levels of each variable in the order `category_lv` gives them. This is the shape an ordered variable such as `high` / `mid` / `low` actually takes, and the one a chi-square test is least efficient at finding, since it spends degrees of freedom on departures that are not there.

A matched design

`paired = TRUE` generates repeated binary measurements on the same subject rather than a cross-classification, so `assoc`, `pattern` and `margins` do not apply and `discordance` takes over. Every subject starts at either level with equal probability, and then moves between consecutive conditions: a subject at the first level takes the second with probability `discordance[1]`, and one at the second takes the first with probability `discordance[2]`.

Two things follow from that, and both are what the matched tests are about. Over two conditions the planted paired odds ratio is exactly `discordance[1] / discordance[2]`, because the equal start makes the two discordant cells half of each transition probability. Over three or more, the response rate climbs from condition to condition whenever the first probability exceeds the second, and that climb is what Cochran's Q is testing for. Equal probabilities are the strict null of both: the transitions cancel, the rate stays at one half, and nothing is planted.

Which arguments each design reads

A cross-classification reads `margins`, `assoc` and `pattern`. A matched design reads `discordance`. Passing one design an argument belonging to the other is a warning naming both lists rather than a value silently ignored, because the two designs plant different things and a call that names the wrong knob is a call that expected the other design.

See Also

`compare_categorical_groups()` for the analysis this feeds, and `simulate_two_groups()` for the numeric counterpart.

Examples

```
## A planted association, recovered by both tests.
sim <- simulate_categorical_groups(n_samples = 300, assoc = 0.4, seed = 1)
fit <- do.call(compare_categorical_groups, sim$args)
fit$tests$chisq_test$pval

## The planted strength beside the estimated one. The estimate is high, which
## is what an association measure built from a sum of squares always is.
cbind(planted = sim$truth$cramers_v,
      estimated = fit$association$estimate[1])

## The residual of each cell against the lift that was planted there.
scored <- merge(fit$cells, sim$truth_cell,
                by = c("row_level", "col_level"))
scored[c("row_level", "col_level", "lift", "std_residual")]

## `assoc = 0` is the product of the margins exactly, so it is null in the
## strict sense: the lift of every cell is 1.
null <- simulate_categorical_groups(n_samples = 300, assoc = 0, seed = 2)
range(null$truth_cell$lift)

## A monotone association over ordered levels, which is the shape a chi-square
## test spends degrees of freedom it does not need on. A 3 x 3 table of this
## many observations is past what Fisher's enumeration can walk, so the
## simulated variant is what answers there.
ordered <- simulate_categorical_groups(
  n_samples   = 300,
  category_lv = list(dose = c("low", "mid", "high"),
                     response = c("none", "partial", "full")),
  assoc       = 0.5,
  pattern     = "gradient",
  seed        = 3
```

```
)
fit_ordered <- do.call(compare_categorical_groups,
                       c(ordered$args, list(simulate_p_value = TRUE)))
as.table(fit_ordered)

## A matched design plants the paired odds ratio as a ratio of transitions,
## and the cells carry what symmetry expects rather than what independence
## would, since symmetry is the null McNemar's test is about.
matched <- simulate_categorical_groups(n_samples = 200, paired = TRUE,
                                       discordance = c(0.3, 0.1), seed = 4)
matched$truth$odds_ratio_paired
matched$truth_cell[c("row_level", "col_level", "p_symmetric",
                     "expected_symmetry_n")]
```

`simulate_classification` *Simulate a two-class outcome whose coefficients are known*

Description

Generates a two-class outcome from a logistic model of its predictors and returns the coefficients that were planted alongside the data, so that a fitted model can be scored against what was actually there. The same design arguments `simulate_regression()` takes are available, together with the class balance, which is the reason a split of a classification has to be stratified.

Usage

```
simulate_classification(
  n_samples = 200,
  n_pred = 8,
  n_pos = round(0.25 * n_pred),
  n_neg = round(0.25 * n_pred),
  beta = NULL,
  beta_range = c(0.5, 2),
  event_rate = 0.3,
  outcome_lv = c("control", "case"),
  value_mean = 0,
  value_sd = 1,
  cor_mat = NULL,
  n_factor_pred = 1,
  factor_lv = c("low", "mid", "high"),
  n_constant_pred = 0,
  p_missing = 0,
  n_per_subject = NULL,
  subject_sd = 1,
  subject_share = 0.5,
  pred_prefix = "x",
```

```
  seed = NULL
)
```

Arguments

n_samples	Rows to generate. Ignored when n_per_subject gives a row count per subject, since those already say how many rows there are.
n_pred	Number of numeric predictors. Columns are named x_1 upwards, or whatever pred_prefix asks for.
n_pos, n_neg	How many numeric predictors are given a positive and a negative coefficient. Their sum cannot exceed n_pred, and every other numeric predictor is left with a coefficient of exactly zero. The defaults are a fraction of n_pred rather than a fixed count, so that asking for fewer predictors plants fewer coefficients instead of failing.
beta	The coefficients themselves, one per numeric predictor and no intercept among them, or NULL to plant n_pos and n_neg of them. Its length is then how many numeric predictors there are, so n_pred need not be given as well; naming both and disagreeing is an error rather than a guess. Supplying it together with n_pos or n_neg is refused, since the two are different ways of saying the same thing.
beta_range	Range the magnitude of a planted coefficient is drawn from. The offsets of the factor predictors are drawn from it too, so it is read whether or not beta was given.
event_rate	Proportion of rows in outcome_lv[2], the class being modelled. The default is deliberately away from a half, since a balanced outcome makes split_data()'s stratification look unnecessary.
outcome_lv	The two class labels, the reference first, so that the coefficients describe the odds of the second one.
value_mean, value_sd	Mean and standard deviation of each numeric predictor, given once for all of them or once each. A coefficient is a change in the outcome per unit of its predictor, so these fix what beta_range means; that is why they are given rather than drawn from a range as the expression simulators draw their spreads.
cor_mat	Correlation matrix of the numeric predictors, as built by make_block_cor(), or NULL to leave them independent. A null predictor correlated with a planted one is the case a coefficient table gets wrong no matter how many rows it is given.
n_factor_pred	Number of categorical predictors, named x_cat_1 upwards. Each becomes length(factor_lv) - 1 terms in the model rather than one, which is why truth_term exists beside truth. Levels are handed out in balanced counts, and each level beyond the first carries a planted offset.
factor_lv	Levels of each categorical predictor, the first being the reference that carries no offset.
n_constant_pred	Number of predictors that take a single value, named x_const_1 upwards. They cannot contribute, so fit_linear_regression() leaves them out and names them in design$dropped_predictors. The default plants none.

`p_missing` Proportion of numeric predictor cells to replace with `NA`, drawn after the outcome has been computed from the complete values. The rows they fall in are the ones a model drops before its folds are laid out, and `design$n_dropped` counts them. Only the numeric predictors are holed, since a hole in the categorical one would stop it being able to stratify a split and a hole in a constant one would stop it being constant.

`n_per_subject` Rows measured on each subject, one entry per subject, so that its length is how many subjects there are. A single number is spread over `n_samples` rows and must divide them. `NULL`, the default, gives no `subject` column. With subjects the class is drawn once per subject from the mean of its rows' probabilities, so a subject is a case or a control as a whole and the outcome can still stratify a split taken over subjects.

`subject_sd` Standard deviation of the per-subject offset on the outcome, which is variation no predictor accounts for. Ignored without `n_per_subject`.

`subject_share` Share of each numeric predictor's variance that lies between subjects rather than within one, so its intraclass correlation. This is what makes two rows of one subject resemble each other, and it is that resemblance a row-wise split gives away: at `0` the rows of a subject are independent draws that happen to share an outcome offset, and no model could tell one subject's rows from another's. How much a model gains from the resemblance depends on the model, and a regression with a fixed set of coefficients gains little, so the argument is what makes the data set the shape `id` is for rather than a way to inflate a score. The distribution of each column is the same whatever it is set to. Ignored without `n_per_subject`.

`pred_prefix` Prefix for the generated predictor names. `"x"` gives `x_1`, `x_cat_1` and `x_const_1`.

`seed` Seed for the draw, or `NULL` to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

A classification differs from a regression in what can go wrong with it, and the defaults are set so that all of it can be seen. Classes are imbalanced, so an unstratified split can hand a fold too few events to fit on. A subject is a case or a control as a whole, so a split that does not respect `id` scores the model on rows whose label it already holds. And a predictor with a coefficient of exactly zero is null in the strict sense, so an odds ratio away from 1 on one is a mistake by definition rather than by judgement.

The linear predictor is `intercept + sum(beta * x) + factor offsets + subject offset`, the class probability is its logistic transform, and the class is a Bernoulli draw from that probability. There is no noise argument: the draw is the noise, which is why a logistic regression recovers less from the same number of rows than a linear one does.

The intercept is not an argument. It is solved for so that the mean class probability over the rows that were actually drawn equals `event_rate`, which means the balance of the data is what was asked for rather than whatever the coefficients happened to imply. `truth_model$intercept` reports the value it took and `truth_model$achieved_event_rate` the proportion the Bernoulli draw then produced.

`outcome_lv` fixes the direction by the rule the rest of the package follows: the first level is the reference, so a planted positive coefficient raises the chance of `outcome_lv[2]` and its odds ratio comes

back above 1. It is carried in `args` rather than left out, because `fit_logistic_regression()` sorts the classes when it is not told them, and `sort(c("case", "control"))` puts `case` first, which would report the odds of the wrong class.

Value

A list of six elements, the same shape `simulate_regression()` returns, with these differences:

`args` Also carries `outcome_lv`, since the direction of every coefficient depends on it and the default would sort the labels the other way round.

`split_args` `stratified` is always the outcome. Unlike a continuous one it is constant within a subject, so it stratifies a split over subjects as readily as one over rows.

`truth_model` `intercept` as solved, the `event_rate` asked for and the `achieved_event_rate` the draw produced, `signal_var`, `subject_var`, `n_samples`, `n_subject` and `subject_sd`. There is no `r_squared`: the outcome is a draw, so no share of its variance is recoverable in that sense.

`truth_row` `prob`, the class probability of the row, and `draw_prob`, the probability the Bernoulli draw actually used, which is the subject's mean when there are subjects and `prob` itself when there are not.

What the defaults are tuned for

The same design `simulate_regression()` uses — eight numeric predictors, four of them planted, one categorical predictor and 200 rows — at an event rate of 0.3, so that the two simulators differ in the outcome and in nothing else. Averaged over twenty seeds that recovers 0.83 of the planted coefficients at `p <= 0.05`, the difficulty the rest of the simulators in this package are tuned to. Coefficients of 0.5 to 2 on the log odds scale are odds ratios of roughly 1.6 to 7, which is what it takes for a class label to carry as much as a number does: the Bernoulli draw is this model's noise and there is no argument to turn it down.

See Also

`fit_logistic_regression()`, which consumes `args` directly, `simulate_regression()` for a continuous outcome, `split_data()`, which consumes `split_args`, and `make_block_cor()` for `cor_mat`.

Examples

```
sim <- simulate_classification(seed = 1)
table(sim$args$data$y)

## The names in `args` are fit_logistic_regression()'s own, `outcome_lv`
## included, so the fit is one call away and points the way it was planted.
fit <- do.call(fit_logistic_regression, c(sim$args, cv = FALSE))

## A planted positive coefficient raises the chance of the second level, so
## its odds ratio is above 1. `truth_term` is in the row order the table uses.
scored <- merge(fit$coefficients, sim$truth_term, by = "terms")
scored[, c("terms", "beta", "estimate", "odds_ratio", "pval")]
table(planted = scored$beta != 0, called = scored$pval <= 0.05)
```

```
## The intercept was not asked for, it was solved for: the balance of the data
## is the balance that was requested.
c(asked = sim$truth_model$event_rate,
  drawn = sim$truth_model$achieved_event_rate)

## Two samples per subject, and a subject is a case or a control as a whole.
## The outcome can therefore still be the stratifier of a split over subjects.
rep_sim <- simulate_classification(n_per_subject = rep(2, 100), seed = 2)
sp <- do.call(split_data, c(rep_sim$split_args, seed = 1))
sp
```

`simulate_factorial_groups` *Simulate a crossed-factor experiment whose answer is known*

Description

The factorial counterpart of `simulate_multiple_groups()`. Crosses any number of factors, lets each one be measured between subjects or within them, and returns the planted answer alongside the data so that a two-way or an n-way analysis can be scored against what was actually put there.

Usage

```
simulate_factorial_groups(
  n_feats = 100,
  factor_lv = list(treatment = c("control", "treat_A", "treat_B", "treat_C"), sex =
    c("male", "female")),
  within = NULL,
  n_per_cell = 20,
  n_up = round(0.15 * n_feats),
  n_down = round(0.15 * n_feats),
  term_mix = c(main_only = 1, additive = 1, interaction = 1, crossover = 1, nuisance_only
    = 1),
  pattern_mix = c(all = 1, gradient = 1, single = 1),
  expr_range = c(2, 12),
  ref_sd = c(1.2, 2.4),
  cell_sd = c(1.8, 3.2),
  deg_log2fc = c(1, 2.5),
  interaction_scale = 0.8,
  subject_sd = c(2, 4),
  feat_prefix = "prot",
  seed = NULL
)
```

Arguments

`n_feats`	Number of features to generate. Columns are named `prot_1` upwards, or whatever `feat_prefix` asks for.
`factor_lv`	Named list of factors, each entry the levels of one factor with the reference level first. Its length is how many factors are crossed, so there is no separate argument for that, and there have to be at least two: one factor is `simulate_multiple_groups()`. The **first factor is the primary one**, the treatment the experiment is about, and the ones after it are the other factors the effect may or may not depend on. The shapes below are written in those terms. The cell in which every factor sits at its reference level is the reference cell, and it is what the planted effects are measured against.
`within`	Names of the factors measured within subjects, or `NULL` for a design that is entirely between them. A subject belongs to one combination of the between factors and is measured under every combination of the within ones, so naming no factor gives a factorial ANOVA, naming all of them gives a fully repeated design, and naming some of them gives a mixed one. When any factor is named, `args` gains `id` and `within`.
`n_per_cell`	Observations per cell. When there are within factors this is also the number of subjects per combination of the between factors, because each of those subjects contributes exactly one observation to each cell it is measured in. One number spreads over every cell; a vector carries one size per combination of the between factors, which makes its length how many of those there are, the rule `n_treat` follows in `simulate_multiple_groups()`. The within factors are fully crossed with every subject, so they cannot hold different sizes.
`n_up`, `n_down`	How many features are moved up and down. Their sum cannot exceed `n_feats`, and every other feature is left with a true effect of exactly zero in every cell and every term. The defaults are a fraction of `n_feats` rather than a fixed count, so that asking for fewer features plants fewer effects instead of failing.
`term_mix`	Named vector of relative weights over the five shapes of effect described under "The five shapes" below, which decide **which terms** of the model an effect is planted in. Set a weight to zero to leave that shape out. The planted features are split between the shapes by the largest remainder method rather than drawn at random, so the counts are exactly what the weights ask for and do not move with the seed.
`pattern_mix`	Named vector of relative weights over `"all"`, `"gradient"` and `"single"`, which decide **how a factor spreads its main effect** over its non-reference levels, exactly as in `simulate_multiple_groups()`. The two mixes are on different axes and are crossed at random: `term_mix` says which terms move and `pattern_mix` says what the profile along a factor looks like. Split by largest remainder too, so both sets of counts are a function of the arguments alone.
`expr_range`	Range the baseline log2 abundance of each feature is drawn from. Every cell shares the baseline, which is what makes an unplanted feature null in every term at once.
`ref_sd`, `cell_sd`	Ranges the per-feature standard deviation of the reference cell and of every other cell are drawn from. Every cell draws its own, so the design is heteroscedastic

and unbalanced variance is something the analysis has to survive rather than something it is spared. Pass the same range twice for equal variances. The defaults are the two ranges `simulate_multiple_groups()` uses.

Keeping them apart costs something that is worth knowing about. An `aov()` interaction test on a `"main_only"` feature, whose interaction is exactly zero, is called about eight times in a hundred rather than five, because the cells whose means differ are also the cells whose spread differs. Passing the same range twice brings it back to about three. The anticonservatism is the test's rather than the simulation's, and finding it is the sort of thing a simulation with a known answer is for.

`deg_log2fc` Range the magnitude of the planted effect is drawn from, on the log2 scale. One magnitude is drawn per planted feature and the shape decides how it is spread over the terms.

`interaction_scale`
Size of the interaction relative to the main effect under the `"interaction"` shape, as a fraction of the magnitude drawn from `deg_log2fc`. It has no bearing on `"crossover"`, where the interaction carries the whole effect because there is nothing else to carry it. Must be above zero: a scale of zero would leave a feature whose `pattern` says `"interaction"` with no interaction in it.

The default puts the `"interaction"` shape between the other two things an interaction row can be. An `aov()` on the defaults finds its interaction about three times in five, against about nine in ten for the `"crossover"` shape and about one in twenty where no interaction was planted, so the row is neither trivially recovered nor indistinguishable from noise. Raising it to 1 takes the shape to nine in ten as well and the two stop differing.

`subject_sd` Range the per-feature subject standard deviation is drawn from. A subject's offset is drawn once per feature and reused across every condition it is measured under, which is what a within-subject test exists to remove. Ignored when `within` is `NULL`.

`feat_prefix` Prefix for the generated feature names. `"prot"` gives `prot_1`, `prot_2` and so on.

`seed` Seed for the draw, or `NULL` to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

One factor asks one question: are the levels alike. Crossing a second one asks three, and they fail separately and for different reasons. Each factor has a main effect, the pair has an interaction, and a design that is read as though the second factor were not there answers none of them. So the effect is planted in five shapes rather than one, chosen to make the three questions come apart: a shape whose main effects are real and whose interaction is not, and a shape whose interaction is real and whose main effects are exactly zero, are both here, and no single test tells them apart.

Value

A list of five elements.

`args` `data`, `feats`, `factors`, `factor_lv` and `input_scale`, and under a within design also `within` and `id`. `factors` is a named list holding one vector per factor, each as long as `data` has rows, and `factor_lv` is the level order of each, exactly as it was passed in. These are the names the factorial comparison will take, so it will be one `do.call()` away when it exists.

`truth` One row per feature, aligned with `feats`, holding `features`, `pattern`, `spread`, `direction`, `partner`, `extreme_cell`, `extreme_tied`, `log2fc`, `baseline` and `sd_subject`.

`truth_term` One row per feature and model term, every main effect and every interaction of every order, holding `features`, `terms`, `term_order`, `is_within`, `max_abs_delta` and `is_effect`. This is the table that scores an ANOVA table row by row, and the one that has no counterpart in `simulate_multiple_groups()`.

`truth_cell` One row per feature and cell, holding `features`, one column per factor, then `is_ref`, `delta`, `center`, `sd` and `n`. A feature the analysis missed can be looked up here rather than guessed at: a large `sd` explains a miss that the effect size alone does not.

`truth_contrast` One row per feature and pair of levels, in the row order and direction a post-hoc table uses, holding `features`, `factor`, `stratum`, `contrast`, `group1`, `group2`, `delta` and `is_diff`. A `stratum` of `NA` is the marginal contrast, averaged over the other factors; anything else names the combination of the other factors the contrast was taken inside, which is the simple effect.

The five shapes

Each planted feature is given a magnitude `d` drawn from `deg_log2fc`, positive for an up feature and negative for a down one, a shape from `term_mix`, and for every shape but the first a partner factor drawn at random from the factors after the primary one. The shape decides which terms of the model end up carrying `d`.

`"main_only"` The primary factor moves and nothing else does. Every other main effect and every interaction is exactly zero. This is `simulate_multiple_groups()` inside a factorial frame, and the case a two-way analysis should answer with one row.

`"additive"` The primary factor and the partner each move, and their interaction is exactly zero. The cell means are the sum of the two, so the profiles are parallel, and an interaction reported as significant here is a false positive by construction.

`"interaction"` The primary factor moves and the size of its effect depends on the level of the partner. The primary main effect and the interaction are both real; the partner's own main effect is left at exactly zero, so the two terms that should be called are the only two there are.

`"crossover"` Pure interaction. The primary factor rises at the partner's reference level and falls at the others, by amounts arranged so that **both main effects are exactly zero** while the cells plainly differ. This is the shape a main-effect test has to miss and an interaction test has to catch, which is the reason a factorial design is analysed as one.

`"nuisance_only"` The partner moves and the primary factor is exactly zero. Read as one factor, the primary factor looks null with inflated within-group spread; read as a factorial design, the spread is accounted for and belongs to a term of its own.

A feature that was not planted has a delta of exactly zero in every cell and a component of exactly zero in every term. Both kinds of mistake are therefore defined for every row of `truth_term`: a term called significant on a zero component is a false positive, and a non-zero component that was not called is a miss.

What the defaults leave recoverable

The defaults are set so that an analysis gets most of the answer and not all of it, which is the band between a simulation that is trivially recovered and one that looks broken. An `aov(y ~ treatment * sex)` on the default 4 by 2 design finds the treatment main effect about four times in five, which is the rate `simulate_multiple_groups()` was tuned to, the two-level factor's main effect rather more often than that, and the interaction between the two, depending on which shape planted it. Every term that was not planted is called at about a twentieth, which is what makes a false positive rate readable off this simulation at all.

The rates are a function of `n_per_cell` as much as of the effect sizes, and they fall away quickly below the default: at eight per cell the `"interaction"` shape's interaction is no longer distinguishable from a null term. A design small enough to be quick is not a design these numbers describe.

How the effect is planted, and how it is reported

The effect is built in the space an ANOVA decomposes into: a main effect is a profile along its own factor that sums to zero, and an interaction sums to zero along each of its factors. The components are added up and the value at the reference cell is then subtracted from the whole array, which leaves the reference cell at exactly zero delta without touching any term, since a constant belongs to the grand mean alone.

That is why the two tables read differently and both are right. `truth_cell$delta` is the shift of a cell from the reference cell, the way `truth_group$delta` is in `simulate_multiple_groups()`, and for a `"main_only"` feature the cell at level `j` of the primary factor carries exactly what the `pattern_mix` profile put there. `truth_term$max_abs_delta` is the largest component of the ANOVA effect itself, which is the quantity that is exactly zero for a term that was not planted. Components smaller than `1e-8` in absolute value are recorded as exactly zero: they are the rounding left over from averaging, and a term left holding `3e-17` would score every row of an ANOVA table against the wrong answer.

Directions

`direction` is the sign of `d`, and it is the sign of the primary factor's effect at the reference level of the partner. `truth$log2fc` is the delta of whichever cell sits furthest from the reference cell, so for every shape but `"crossover"` an up feature is positive there. Under `"crossover"` the primary factor moves in opposite directions at different levels of the partner, so which cell is furthest, and its sign, follow from the shape rather than from `direction`.

`truth_contrast$delta` is `group1 - group2` with `group1` the later level of the factor, which is the direction and the row order a post-hoc table uses. It comes from `sa_level_pairs()`, the same helper the post-hoc tables are built from, so the two cannot drift apart.

`extreme_cell` is the levels of that cell joined by a dot, so it reads back against `truth_cell` without a lookup. When more than one cell is equally far from the reference, which is what the `"all"` profile does on purpose, it records the first of them and `extreme_tied` is `TRUE`: the flag that says to score the magnitude rather than the name of the cell. It is `TRUE` with `extreme_cell` missing for an unplanted feature, whose cells are all zero and none of which is furthest.

Within and between

A subject belongs to one combination of the between factors and is measured under every combination of the within ones, so no subject is dropped and the within-subject rectangle is complete. Each

subject gets an offset per feature, drawn once and added to all of its rows, which is the between-subject variation a within-subject test removes. The residual standard deviation still differs from cell to cell, so sphericity does not hold and the corrections a repeated measures analysis reports have something to report.

See Also

simulate_multiple_groups() for the one-factor case, whose pattern_mix shapes this reuses, and simulate_two_groups() for two groups.

Examples

```
## A 4 x 2 design: four treatments crossed with sex, both between subjects.
sim <- simulate_factorial_groups(n_feats = 20, n_up = 5, n_down = 5,
                                 n_per_cell = 6, seed = 1)
table(pattern = sim$truth$pattern, direction = sim$truth$direction)

## The answer per term. A "crossover" feature has an interaction and no main
## effect at all, which is what makes the shape worth planting.
cross <- sim$truth$features[sim$truth$pattern == "crossover"][1]
subset(sim$truth_term, features == cross,
       select = c("terms", "max_abs_delta", "is_effect"))

## Scored against a two-way ANOVA. The term names line up with `truth_term`,
## so the two tables merge without being renamed.
long <- data.frame(y = sim$args$data[[cross]], sim$args$factors)
summary(stats::aov(y ~ treatment * sex, data = long))

## An "additive" feature has both main effects and no interaction, so it is
## the shape that makes an interaction call a false positive.
add <- sim$truth$features[sim$truth$pattern == "additive"][1]
subset(sim$truth_term, features == add, select = c("terms", "is_effect"))

## A missed feature is looked up rather than guessed at. The row per cell
## carries the spread the cell was given, which is one of the reasons.
head(subset(sim$truth_cell, features == cross), 4)

## Marginal contrasts and simple effects are both in `truth_contrast`. A
## stratum of NA is the marginal one.
head(subset(sim$truth_contrast, features == cross & factor == "treatment"), 3)

## Time measured on the same subjects, treatment and sex between them: a
## mixed design. `args` then carries `id` and `within`.
mixed <- simulate_factorial_groups(
  n_feats = 10, n_up = 2, n_down = 2, n_per_cell = 4,
  factor_lv = list(treatment = c("control", "treat_A", "treat_B"),
                   sex       = c("male", "female"),
                   time      = c("T0", "T1", "T2")),
  within = "time", seed = 1
)
names(mixed$args)
## Every subject under every time point, so the rectangle is complete.
```

```
table(table(mixed$args$id))

## Which terms are tested in the within-subject error stratum is recorded, so
## the answer table knows which half of a mixed ANOVA to score.
unique(mixed$truth_term[c("terms", "term_order", "is_within")])
```

`simulate_multiple_groups` *Simulate a control-versus-treatments experiment whose answer is known*

Description

The multi-group counterpart of `simulate_two_groups()`. Generates log2-scale abundance data for one control group and any number of treatment groups, and returns the planted answer alongside the data so that a comparison can be scored against what was actually put there.

Usage

```
simulate_multiple_groups(
  n_feats = 100,
  n_control = 50,
  n_treat = rep(50, 3),
  n_up = round(0.15 * n_feats),
  n_down = round(0.15 * n_feats),
  pattern_mix = c(all = 1, gradient = 1, single = 1),
  expr_range = c(2, 12),
  control_sd = c(1.2, 2.4),
  treat_sd = c(1.8, 3.2),
  deg_log2fc = c(1, 2.5),
  paired = FALSE,
  subject_sd = c(2, 4),
  group_lv = NULL,
  feat_prefix = "prot",
  seed = NULL
)
```

Arguments

`n_feats`	Number of features to generate. Columns are named `prot_1` upwards, or whatever `feat_prefix` asks for.
`n_control`	Number of observations in the control group.
`n_treat`	Observations in each treatment group, one entry per group, so that its length is how many treatment groups there are. Pass `c(50, 40, 30)` for three treatment groups of different sizes and `rep(30, 5)` for five of the same size. There is no

separate argument for the number of groups: the sizes already say it, and two arguments that could disagree would have to be settled somewhere.

compare_multiple_groups() needs at least three levels in all, so this needs at least two entries. A single number is allowed only when group_lv says how many groups to spread it over.

n_up, n_down
: How many features are moved up and down in the treatment groups. Their sum cannot exceed n_feats, and every other feature is left with a true effect of exactly zero in every group. The defaults are a fraction of n_feats rather than a fixed count, so that asking for fewer features plants fewer effects instead of failing; at the default n_feats = 100 they are the 15 and 15 the defaults were tuned at.

pattern_mix
: Named vector of relative weights over the three shapes an effect can take, described under "The three shapes" below. Set a weight to zero to leave that shape out. The planted features are split between the shapes in these proportions by the largest remainder method rather than drawn at random, so the counts are exactly what the weights ask for and do not move with the seed.

expr_range
: Range the baseline log2 abundance of each feature is drawn from. Every level shares the baseline, which is what makes an unplanted feature null.

control_sd, treat_sd
: Ranges the per-feature standard deviation of the control group and of each treatment group are drawn from. Every group draws its own, so the design is heteroscedastic, which is the situation Welch's ANOVA and Games-Howell exist for. Pass the same range twice for equal variances. The defaults are the two ranges simulate_two_groups() uses and leave roughly four planted features in five recoverable by the omnibus test at the default cutoffs; narrowing them recovers nearly everything and widening them costs recall quickly.

deg_log2fc
: Range the magnitude of the planted effect is drawn from, on the log2 scale. This is the magnitude at the level that carries the full effect; the "gradient" shape places the intermediate levels below it.

paired
: Logical. If TRUE, the levels are treated as repeated conditions measured on the same subjects, and args gains id and paired so that compare_multiple_groups() runs the within-subject tests.

subject_sd
: Range the per-feature subject standard deviation is drawn from. A subject's offset is drawn once per feature and reused across every condition, which is what a within-subject test exists to remove. The default is deliberately of the same order as the residual spread, so that analysing the same table without id costs most of the recall. A small offset would let the unpaired analysis do nearly as well and would teach the opposite of why the design exists. Ignored when paired = FALSE.

group_lv
: Group labels, the first being the control that every effect is planted against. Defaults to c("control", "treat_1", ...), one label per entry of n_treat plus the control. When supplied it says how many groups there are, so a n_treat of length one is spread over them and a n_treat left at its default is replaced by that many groups of the default size. Supplying labels and sizes that count differently is an error rather than a guess.

feat_prefix
: Prefix for the generated feature names. "prot" gives prot_1, prot_2 and so on.

seed — Seed for the draw, or NULL to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

With two groups there is one thing to get right: whether a feature moved. With three or more there are two, and they fail separately. The omnibus test asks whether the levels are all alike, and the post-hoc stage asks which of them differ. A feature can clear the first and be misread by the second, and a shape of effect that the omnibus finds easy can be the one the post-hoc stage finds hard. That is why the effect is planted in three shapes rather than one, and why the answer comes back in three tables rather than one.

Value

A list of four elements.

args data, feats, group, group_lv and input_scale, named after the arguments of compare_multiple_groups() so that do.call(compare_multiple_groups, sim$args) runs the comparison. Under paired = TRUE it also carries id and paired.

truth One row per feature, aligned with feats, holding features, pattern, direction, extreme_level, extreme_tied, log2fc, baseline and sd_subject. This is the table that scores $effect and the omnibus tests.

truth_group One row per feature and level, holding features, group, is_ref, delta, center, sd and n. A feature the comparison missed can be looked up here rather than guessed at: a large sd explains a miss that the effect size alone does not.

truth_contrast One row per feature and pair of levels, in the row order and direction the post-hoc tables use, holding features, contrast, group1, group2, delta and is_diff. This is the table that scores $posthoc.

The three shapes

Each planted feature is given a magnitude d drawn from deg_log2fc, positive for an up feature and negative for a down one, and one of three shapes that decides what each treatment group does with it.

"all" Every treatment group is shifted by d. Only the control stands apart, so the omnibus test has the whole effect to work with and every contrast against the control should be found.

"gradient" Treatment group g of k is shifted by d * g / k, so the last one carries the full effect and the ones before it carry a fraction. This is the dose-response shape, and its early contrasts are the ones a post-hoc stage loses first.

"single" One treatment group, chosen at random, is shifted by d and the rest are left at exactly zero. The omnibus test is diluted here, since most of the levels it compares are alike, so this is the shape it misses most often. When it does clear the cutoff, exactly the contrasts involving that one level should come back.

A feature that was not planted has a delta of exactly zero in every group. Both kinds of mistake are therefore defined: a contrast called significant on a zero delta is a false positive, and a non-zero delta that was not called is a miss.

Directions

`truth$log2fc` is the delta of whichever level sits furthest from the control, which is the quantity `$effect$log2fc` estimates. A treatment group that went up gives a positive value in both.

`truth_contrast$delta` reads the same way, because the post-hoc tables do. A post-hoc `estimate` is `group1 - group2` with `group1` the later level of `group_lv`, and the control is the first level, so the contrast is `treat_1 - control` and a feature whose treatment groups went up is positive there too. `truth_contrast$delta` is computed as the same difference, so every direction in the three tables points one way.

Under the `"all"` shape every treatment group carries the same delta, so no single level is furthest from the control. `extreme_level` then records the first of the tied levels and `extreme_tied` is `TRUE`, which is the flag that says to score the magnitude rather than the name of the level. It is also `TRUE`, with `extreme_level` missing, for an unplanted feature.

Repeated conditions

Under `paired = TRUE` each subject is measured under every condition, so no subject is dropped and `design$unmatched_ids` comes back empty. Each subject gets an offset per feature, drawn once and added to all of its conditions, which is the between-subject variation the within-subject tests remove. The residual standard deviation still differs between conditions, so sphericity does not hold and the Mauchly, Greenhouse-Geisser and Huynh-Feldt columns of the repeated measures ANOVA have something to report.

The same subjects appearing under every condition also means every group holds the same number of them, so `n_control` and every entry of `n_treat` have to agree. Unequal sizes are rejected rather than quietly levelled, since the sizes are the clearest statement of which design was meant.

See Also

`compare_multiple_groups()`, which consumes `args` directly, `simulate_two_groups()` for the two-group case, and `estimate_significance()` for the verdict that `truth` is there to score.

Examples

```
sim <- simulate_multiple_groups(n_feats = 30, n_up = 5, n_down = 5, seed = 1)
table(pattern = sim$truth$pattern, direction = sim$truth$direction)

## The names in `args` are compare_multiple_groups()'s own, so the comparison
## is one call away.
res <- do.call(compare_multiple_groups, c(sim$args, diagnose = FALSE))
sig <- estimate_significance(res, test = "anova_test")$significance

## Scored against what was planted. The off-diagonal cells are the two kinds
## of mistake: features that were planted and missed, and null features that
## were called anyway.
planted <- sim$truth$direction != "none"
table(planted = planted, called = sig$is_signif %in% TRUE)

## The shape of the effect decides how hard the omnibus test finds it. A
## "single" feature differs from the control in one level out of several, so
## most of what the test compares is alike.
```

```
tapply(sig$is_signif %in% TRUE, sim$truth$pattern, mean)

## The pairwise stage is scored against `truth_contrast`, which is already in
## the row order and the direction the post-hoc table uses.
ph <- merge(res$posthoc$anova_test, sim$truth_contrast,
            by = c("features", "contrast"))
table(differs = ph$is_diff, called = ph$pval_adj <= 0.05)

## A missed feature is looked up rather than guessed at. The row per level
## carries the size the group was given, which is one of the reasons.
subset(sim$truth_group, features == sim$truth$features[1])

## `n_treat` is one size per treatment group, so its length is how many there
## are. Five groups of unequal size, against a control of 40:
uneven <- simulate_multiple_groups(n_feats = 20, n_control = 40,
                                   n_treat = c(30, 25, 20, 15, 10), seed = 1)
table(uneven$args$group)[uneven$args$group_lv]

## Labels alone also say how many groups there are, so one size is spread
## over them.
dose <- simulate_multiple_groups(n_feats = 20, n_treat = 25,
                                 group_lv = c("dmso", "low", "mid", "high"),
                                 seed = 1)
table(dose$args$group)[dose$args$group_lv]

## Repeated conditions: the same subjects under every treatment, so every
## group holds the same number of them.
rep_sim <- simulate_multiple_groups(n_feats = 10, n_up = 2, n_down = 2,
                                    n_control = 12, n_treat = rep(12, 3),
                                    paired = TRUE, seed = 1)
rep_res <- do.call(compare_multiple_groups, c(rep_sim$args, diagnose = FALSE))
rep_res$tests$anova_test[1:3, c("features", "f_stat", "pval_adj")]
```

`simulate_regression` *Simulate a regression whose coefficients are known*

Description

Generates a continuous outcome from a linear combination of predictors and returns the coefficients that were planted alongside the data, so that a fitted model can be scored against what was actually there. Everything a model has to survive can be asked for: predictors that correlate, a categorical predictor, a predictor that takes one value, missing cells, and repeated measurements of the same subject.

Usage

```
simulate_regression(
  n_samples = 200,
  n_pred = 8,
```

```
  n_pos = round(0.25 * n_pred),
  n_neg = round(0.25 * n_pred),
  beta = NULL,
  beta_range = c(0.5, 2),
  intercept = 0,
  value_mean = 0,
  value_sd = 1,
  noise_sd = 3,
  cor_mat = NULL,
  n_factor_pred = 1,
  factor_lv = c("low", "mid", "high"),
  n_constant_pred = 0,
  p_missing = 0,
  n_per_subject = NULL,
  subject_sd = 1,
  subject_share = 0.5,
  pred_prefix = "x",
  seed = NULL
)
```

Arguments

`n_samples`	Rows to generate. Ignored when `n_per_subject` gives a row count per subject, since those already say how many rows there are.
`n_pred`	Number of numeric predictors. Columns are named `x_1` upwards, or whatever `pred_prefix` asks for.
`n_pos`, `n_neg`	How many numeric predictors are given a positive and a negative coefficient. Their sum cannot exceed `n_pred`, and every other numeric predictor is left with a coefficient of exactly zero. The defaults are a fraction of `n_pred` rather than a fixed count, so that asking for fewer predictors plants fewer coefficients instead of failing.
`beta`	The coefficients themselves, one per numeric predictor and no intercept among them, or `NULL` to plant `n_pos` and `n_neg` of them. Its length is then how many numeric predictors there are, so `n_pred` need not be given as well; naming both and disagreeing is an error rather than a guess. Supplying it together with `n_pos` or `n_neg` is refused, since the two are different ways of saying the same thing.
`beta_range`	Range the magnitude of a planted coefficient is drawn from. The offsets of the factor predictors are drawn from it too, so it is read whether or not `beta` was given.
`intercept`	The intercept. It is not part of `beta`.
`value_mean`, `value_sd`	Mean and standard deviation of each numeric predictor, given once for all of them or once each. A coefficient is a change in the outcome per unit of its predictor, so these fix what `beta_range` means; that is why they are given rather than drawn from a range as the expression simulators draw their spreads.
`noise_sd`	Standard deviation of the residual noise. This and `beta_range` together decide how much of the outcome is recoverable at all; `truth_model$r_squared` reports how much, on the design that was drawn.

cor_mat
: Correlation matrix of the numeric predictors, as built by make_block_cor(), or NULL to leave them independent. A null predictor correlated with a planted one is the case a coefficient table gets wrong no matter how many rows it is given.

n_factor_pred
: Number of categorical predictors, named x_cat_1 upwards. Each becomes length(factor_lv) - 1 terms in the model rather than one, which is why truth_term exists beside truth. Levels are handed out in balanced counts, and each level beyond the first carries a planted offset.

factor_lv
: Levels of each categorical predictor, the first being the reference that carries no offset.

n_constant_pred
: Number of predictors that take a single value, named x_const_1 upwards. They cannot contribute, so fit_linear_regression() leaves them out and names them in design$dropped_predictors. The default plants none.

p_missing
: Proportion of numeric predictor cells to replace with NA, drawn after the outcome has been computed from the complete values. The rows they fall in are the ones a model drops before its folds are laid out, and design$n_dropped counts them. Only the numeric predictors are holed, since a hole in the categorical one would stop it being able to stratify a split and a hole in a constant one would stop it being constant.

n_per_subject
: Rows measured on each subject, one entry per subject, so that its length is how many subjects there are. A single number is spread over n_samples rows and must divide them. NULL, the default, gives no subject column and one row per sampling unit. This is what split_data()'s id argument exists for: a subject partly seen in training is partly known before its test rows are read.

subject_sd
: Standard deviation of the per-subject offset on the outcome, which is variation no predictor accounts for. Ignored without n_per_subject.

subject_share
: Share of each numeric predictor's variance that lies between subjects rather than within one, so its intraclass correlation. This is what makes two rows of one subject resemble each other, and it is that resemblance a row-wise split gives away: at 0 the rows of a subject are independent draws that happen to share an outcome offset, and no model could tell one subject's rows from another's. How much a model gains from the resemblance depends on the model, and a regression with a fixed set of coefficients gains little, so the argument is what makes the data set the shape id is for rather than a way to inflate a score. The distribution of each column is the same whatever it is set to. Ignored without n_per_subject.

pred_prefix
: Prefix for the generated predictor names. "x" gives x_1, x_cat_1 and x_const_1.

seed
: Seed for the draw, or NULL to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

The point of the exercise is the gap between the coefficient table and the truth, and the three things that open it. A planted coefficient can be too small for the noise to let it through, a null predictor correlated with a planted one is estimated away from zero however much data there is, and a subject

measured repeatedly makes a row-wise split score the model on rows it half knows already. The defaults are set so that a run recovers most but not all of what was planted, because a simulation that recovers everything teaches none of this.

Each row is drawn as `y = intercept + sum(beta * x) + factor offsets + subject offset + noise`, with the numeric predictors drawn from a multivariate normal whose correlations are `cor_mat` and the noise normal with standard deviation `noise_sd`. Because the outcome is built from the coefficients and nothing else, a predictor whose coefficient is zero is null in the strict sense, and a p-value below the cutoff on one is a false positive by definition.

Which predictors carry a planted coefficient is drawn at random, but how many carry a positive one and how many a negative one is not: `n_pos` and `n_neg` are counts, so they do not move with the seed. `beta` states every coefficient instead, in which case nothing is planted and its length is how many numeric predictors there are.

Value

A list of six elements.

`args` `data`, `outcome` and `predictors`, named after the arguments of `fit_linear_regression()` so that `do.call(fit_linear_regression, sim$args)` fits the model. `predictors` is given explicitly rather than left to its `NULL` default, which would take the `subject` column as a predictor and let the model fit on which subject a row came from.

`split_args` `data`, `stratified` and `id`, named after the arguments of `split_data()`. The outcome is the stratifier when there are no subjects; with subjects it varies within a subject and so cannot stratify a split taken over them, and the first categorical predictor, which is drawn per subject, is used instead.

`truth` One row per predictor, in the column order of `data`, holding `predictors`, `role` (`"signal"`, `"null"`, `"factor"` or `"constant"`), `beta`, `direction`, `value_mean`, `value_sd` and `max_cor_signal`, the largest correlation this predictor has with a planted one. The last is what accounts for a null predictor that came back significant.

`truth_term` One row per model term, in the row order `coefficients` follows, holding `terms`, the `predictors` each term came from, and `beta`. This is the table that scores the coefficients, since a categorical predictor is several terms and a constant one is none.

`truth_model` The model as a whole: `intercept`, `noise_sd`, `signal_var`, `subject_var` and `r_squared`, the share of the variance of the outcome the predictors account for, which is what `fit_stats$r_squared` estimates. Also `n_samples`, `n_subject` and `subject_sd`.

`truth_row` One row per observation, holding `subject`, `subject_offset`, `eta` (the whole linear predictor, intercept included) and `noise`, so that `y` is exactly `eta + noise`.

What the defaults are tuned for

Eight numeric predictors, four of them planted, one categorical predictor and 200 rows, with `noise_sd = 3` against coefficients of 0.5 to 2 on unit-variance predictors. Averaged over twenty seeds that leaves 47% of the variance of the outcome accounted for by the predictors and recovers 0.83 of the planted coefficients at `p <= 0.05`, which is the same difficulty `simulate_two_groups()` is tuned to. Lowering `noise_sd` to 2 recovers 0.94 and raising it to 5 costs a third of them.

The rate at which a null predictor is called is 0.05 and stays there, because a coefficient table applies no multiplicity adjustment across its terms. That is a property of the model rather than of

these defaults, and it is one of the things a simulation with strictly null predictors is for: with eight predictors and a cutoff of 0.05, a table that names a predictor it should not is the expected outcome of roughly every other run.

See Also

`fit_linear_regression()`, which consumes `args` directly, `simulate_classification()` for a two-class outcome, `split_data()`, which consumes `split_args`, and `make_block_cor()` for `cor_mat`.

Examples

```
sim <- simulate_regression(seed = 1)
sim$truth[, c("predictors", "role", "beta")]

## The names in `args` are fit_linear_regression()'s own, so the fit is one
## call away.
fit <- do.call(fit_linear_regression, c(sim$args, cv = FALSE))

## Scored against what was planted. `truth_term` is already in the row order
## the coefficient table follows, so the two line up without matching.
scored <- merge(fit$coefficients, sim$truth_term, by = "terms")
scored[, c("terms", "beta", "estimate", "pval")]

## Both kinds of mistake are defined, because a null predictor's coefficient
## is exactly zero rather than merely small.
table(planted = scored$beta != 0, called = scored$pval <= 0.05)

## The model as a whole is scored too: r_squared estimates the share of the
## variance the predictors actually carry.
c(planted = sim$truth_model$r_squared, fitted = fit$fit_stats$r_squared)

## Correlated predictors are the reason a coefficient table names the wrong
## one. `x_2` is null and correlates 0.9 with `x_1`, which is not.
pair <- simulate_regression(
  n_pred = 4, beta = c(2, 0, 0, 0),
  cor_mat = make_block_cor(4, list(list(features = 1:2, cor = 0.9))),
  seed = 2
)
pair$truth[, c("predictors", "role", "beta", "max_cor_signal")]

## Three measurements per subject, so a split has to be taken over subjects.
rep_sim <- simulate_regression(n_per_subject = rep(3, 40), seed = 3)
sp <- do.call(split_data, c(rep_sim$split_args, seed = 1))
intersect(sp$datasets[[1]]$train_data$subject,
          sp$datasets[[1]]$test_data$subject)
```

`simulate_two_groups` *Simulate a two-group experiment whose answer is known*

Description

Generates log2-scale expression data for two independent groups with a fixed number of features moved up and down on purpose, and returns the planted answer alongside the data. Every quantity a comparison estimates can then be checked against what was actually put there, which is what a real data set can never offer.

Usage

```
simulate_two_groups(
  n_feats = 100,
  n_case = 50,
  n_control = 50,
  n_up = 15,
  n_down = 15,
  expr_range = c(2, 12),
  case_sd = c(1.8, 3.2),
  control_sd = c(1.2, 2.4),
  deg_log2fc = c(1, 2.5),
  group_lv = c("control", "case"),
  seed = NULL
)
```

Arguments

`n_feats`	Number of features to generate. Columns are named `gene_1` upwards.
`n_case`, `n_control`	Observations in each group. They do not have to match.
`n_up`, `n_down`	How many features are moved up and down in the case group. Their sum cannot exceed `n_feats`, and every other feature is left with a true fold change of exactly zero.
`expr_range`	Range the baseline log2 expression of each feature is drawn from. The default spans what log2 CPM or RMA values usually cover. Both groups share the baseline, which is what makes an unplanted feature null.
`case_sd`, `control_sd`	Ranges the per-feature standard deviation of each group is drawn from. They are drawn independently, so the groups end up with unequal variances, which is the situation Welch's t-test and the Brunner-Munzel test exist for. Pass the same range twice for a homoscedastic data set. The defaults leave roughly four planted features in five recoverable at the default cutoffs; narrowing them recovers nearly everything and widening them costs recall quickly.

`deg_log2fc` Range the magnitude of the planted effect is drawn from, on the log2 scale. The default of `c(1, 2.5)` is a two-fold to roughly six-fold change, which straddles the `log2fc_cutoff = 1` that `estimate_significance()` applies by default.

`group_lv` The two group labels, the first being the control and the second the one the effect is applied to. Passed straight through to the returned arguments, so it also fixes the direction `compare_two_groups()` reads: a planted increase comes back as a positive `log2fc` because the control is the reference.

`seed` Seed for the draw, or `NULL` to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

The point of the exercise is the gap. A comparison does not recover every feature that was planted, and the reasons it misses them are the three things worth understanding about a volcano plot: the p-value may not clear its cutoff, the multiplicity adjustment may take it back, and the estimated `log2fc` carries a sampling error of its own, so a feature planted just above the magnitude cutoff lands below it about half the time. The defaults are set so that a run recovers most but not all of what was planted, because a simulation that recovers everything teaches none of this.

Value

A list of two elements.

`args` `data`, `feats`, `group`, `group_lv` and `input_scale`, named after the arguments of `compare_two_groups()` so that `do.call(compare_two_groups, sim$args)` runs the comparison. `input_scale` is `"log2"`, since that is the scale the data is on.

`truth` One row per feature, aligned with `feats`, holding `features`, `direction` (`"up"`, `"down"` or `"none"`), `log2fc` (the effect that was planted, exactly `0` for `"none"`), `baseline` and the two group standard deviations. The last three are there so that a feature the comparison missed can be looked up rather than guessed at: a large `sd_case` explains a miss that the effect size alone does not.

How the data is built

Each feature gets a baseline `b` drawn from `expr_range`, two standard deviations drawn from `case_sd` and `control_sd`, and a planted effect `d` that is a positive draw from `deg_log2fc` when the feature is one of the `n_up`, a negative draw when it is one of the `n_down`, and `0` otherwise. Case observations are then normal around `b + d` and control observations normal around `b`.

Because the baseline is shared, the true log2 fold change of a feature is `d` and nothing else. An unplanted feature is null in the strict sense, so a feature called significant is a false positive by definition and the multiplicity adjustment can be judged on it. A model that gave each group its own random offset would look more lifelike and would make both the recall and the false positive rate impossible to compute, since an unplanted feature would then differ between the groups too.

The data is on the log2 scale throughout, so `input_scale = "log2"` comes back with it. The effect is added rather than multiplied, which is what makes `deg_log2fc` a difference of log2 means rather than a ratio.

See Also

compare_two_groups(), which consumes args directly, and estimate_significance() for the verdict that truth is there to score.

Examples

```
sim <- simulate_two_groups(seed = 1)
table(sim$truth$direction)

## The names in `args` are compare_two_groups()'s own, so the comparison is
## one call away.
res <- do.call(compare_two_groups, sim$args)
sig <- estimate_significance(res, test = "t_test")$significance

## Scored against what was planted. The off-diagonal cells are the two kinds
## of mistake: features that were planted and missed, and null features that
## were called anyway.
planted <- sim$truth$direction != "none"
called <- sig$is_signif %in% TRUE
table(planted = planted, called = called)

## The direction is recovered too, not just the fact of a difference.
table(truth = sim$truth$direction[called], sign = sign(sig$log2fc[called]))

## Recall differs between the three families on the same data and the same
## truth, which is the reason all three are reported.
vapply(names(res$tests), function(nm) {
  hit <- estimate_significance(res, test = nm)$significance$is_signif
  mean(hit[planted] %in% TRUE)
}, numeric(1))

## Turning the noise up costs recall without changing what was planted.
noisy <- simulate_two_groups(seed = 1, case_sd = c(4, 6),
                             control_sd = c(4, 6))
noisy_sig <- estimate_significance(
  do.call(compare_two_groups, noisy$args)
)$significance
mean(noisy_sig$is_signif[noisy$truth$direction != "none"] %in% TRUE)
```

---

split_data *Split data into training and test sets*

Description

Partitions the rows of a data set into a training half and a test half, optionally several times over. The partition is stratified, so the balance of the whole data set is preserved in both halves rather than left to the draw, and it can be taken over sampling units rather than over rows, so that repeated measurements of one subject never end up on both sides.

Usage

```
split_data(
  data,
  stratified = NULL,
  id = NULL,
  p_train = 0.75,
  times = 1,
  seed = NULL
)
```

Arguments

`data`	A data.frame (or matrix) in wide format, one row per observation.
`stratified`	What to preserve the balance of, either the name of a column of `data` or a vector with one entry per row. Usually the outcome the model will predict. `NULL` draws a simple random split.
`id`	Sampling unit, either a column name or a vector with one entry per row. Rows sharing a value are assigned to the same side of the split, which is what keeps repeated measurements or technical replicates of one subject from appearing in both halves. `NULL` treats every row as its own unit.
`p_train`	Proportion allocated to the training set, strictly between 0 and 1.
`times`	Number of independent splits to draw. The shape of the result does not depend on it: one split still comes back as a list of one.
`seed`	Seed for the draw, or `NULL` to use the stream as it stands. Supplying one does not disturb the caller: the previous random number state is put back when the function returns.

Details

Nothing is fitted or transformed here. The point of splitting first is that every later step — imputation, scaling, feature selection, hyperparameter tuning — is fitted on the training half alone, and this function is what makes that half well defined.

The partition itself is drawn by `caret::createDataPartition()`, which takes `ceiling(n * p_train)` observations from within each stratum. What it does with `stratified` depends on its type, and the difference is worth knowing because it is invisible from the outside:

Factor or character Each distinct value is a stratum. A value occurring only once is put into the training set and the test set has none of it, which `createDataPartition()` warns about.

Numeric Cut into up to five quantile bins first, and the bins are the strata. This is how a continuous outcome is kept from landing entirely on one side, but it does mean a numeric stratifier is never matched exactly.

`NULL` No strata. The split is a simple random draw of `ceiling(nrow(data) * p_train)` rows.

With `id` the whole thing moves up one level. Rows are folded into units, each unit takes the stratum its rows agree on, the partition is drawn over units, and the chosen units are expanded back into row indices. `p_train` is then a proportion of units, not of rows, and the two differ whenever the units have unequal sizes; `parameters$achieved_p` reports the row proportion each repeat actually reached.

Value

An object of class `sa_split`, a plain list of five elements.

`full_data` The input, exactly as it was passed in. Held once rather than once per repeat.

`datasets` One element per repeat, named `Resample1` upwards, each a list of `train_data`, `test_data`, and the `train_rows` and `test_rows` they were taken from. The two frames have their row names reset, so the row numbers are the only record of where a row came from and they are kept beside it.

`train_idx` The training row indices of every repeat, in the form `caret::createDataPartition(list = TRUE)` returns. A matrix is not offered because units of unequal size make the repeats different lengths.

`design` What the split was made on: row and unit counts, the labels of `stratified` and `id`, and the number of units per stratum where the strata are discrete.

`parameters` `p_train`, `times`, `seed`, and `achieved_p`, the row proportion each repeat actually reached.

See Also

`caret::createDataPartition()`, which draws the partition.

Examples

```
## Stratified on the outcome: both halves keep the species balance of the
## whole data set rather than whatever the draw happened to give.
sp <- split_data(iris, stratified = "Species", seed = 1)
sp
table(sp$datasets[[1]]$train_data$Species)
table(sp$datasets[[1]]$test_data$Species)

## Three splits at once, for a repeated hold-out.
sp3 <- split_data(iris, stratified = "Species", times = 3, seed = 1)
vapply(sp3$datasets, function(d) nrow(d$train_data), numeric(1))

## Three measurements per subject. Splitting by row would put most subjects
## in both halves; splitting by `id` cannot.
rep_data <- data.frame(
  subject = rep(paste0("s", 1:20), each = 3),
  arm     = rep(c("control", "treated"), each = 30),
  value   = seq_len(60)
)
sp_id <- split_data(rep_data, stratified = "arm", id = "subject", seed = 1)
intersect(sp_id$datasets[[1]]$train_data$subject,
          sp_id$datasets[[1]]$test_data$subject)

## `p_train` is a proportion of units once `id` is given, so the row
## proportion it reaches is reported rather than assumed.
sp_id$parameters$achieved_p
```

summarize_association_stats *Correlation between every pair of features, with all three coefficients*

Description

Reduces a set of features to the association between each pair of them, as a square matrix per quantity: the coefficient, its p-value, the p-value adjusted across the pairs, and how many observations the pair shared. Pearson, Spearman and Kendall are reported side by side on the same pairs, the way the comparison functions report a parametric, a rank-based and a robust test side by side, so that a linear coefficient and a monotonic one disagreeing is visible rather than a matter of which call was made.

Usage

```
summarize_association_stats(
  data,
  feats = NULL,
  methods = c("pearson", "spearman", "kendall"),
  adj_type = "BH",
  use = c("pairwise.complete.obs", "complete.obs")
)
```

Arguments

`data`	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
`feats`	Character vector of numeric column names in `data` to correlate, in output order, or `NULL` for every numeric column of `data`. At least two are needed.
`methods`	Which coefficients to compute, drawn from `"pearson"`, `"spearman"` and `"kendall"`. Each one named gets a slot of the result, in the order given; the ones not named have no slot at all.
`adj_type`	Multiplicity adjustment applied across the pairs, any method from stats::p.adjust.methods.
`use`	How a missing value is handled. `"pairwise.complete.obs"` reads each pair on the observations that pair shares, and `"complete.obs"` drops any row with a missing value in any feature first, so that every pair is read on one set of rows.

Details

This is a screen rather than a test of one hypothesis. It is the companion of `summarize_descriptive_stats()`, which reduces one feature at a time to a row of its own; this reduces a pair at a time. Neither returns a `sa_comparison`.

Every matrix is symmetric. The upper triangle is computed and mirrored, so a pair is tested once rather than twice.

The diagonal is a property of the matrix rather than an estimate: `corr` is 1 and `pvalue` and `adj_pvalue` are `NA`, since a feature is not tested against itself. `n` on the diagonal is the number of observations that feature has.

The adjustment runs over the pairs that produced a p-value. A pair `stats::cor.test()` refused, having fewer than three shared observations or no variance on one side, is not a test that was performed, and counting it into the family would shrink the other pairs for a comparison that never happened. Such a pair comes back `NA` in all three of `corr`, `pvalue` and `adj_pvalue` rather than aborting the screen, and features with no variance at all are named in a message.

Missing and non-finite values are treated alike, an `Inf` being as much "no value to correlate" as an `NA`, and `n` counts what was left.

The cost is one `stats::cor.test()` per pair per method: thirty features are 435 pairs, and 1305 tests with all three methods asked for. Kendall's is the slowest of the three, so naming `methods` is worth doing on a wide frame.

Value

A plain list. One slot per entry of `methods`, named after it and holding four features-by-features matrices with the same dimnames:

`corr` The coefficient, from `stats::cor()`.

`pvalue` The two-sided p-value from `stats::cor.test()`.

`adj_pvalue` `pvalue` adjusted by `adj_type` across the pairs.

`n` Observations the pair shared.

Beside those, `design` records `feats`, `n_obs`, `methods`, `adj_type` and `use`.

See Also

`draw_corrplot()` to draw a method's `corr` with the non-significant cells left blank, and `summarize_descriptive_stats()` for the one-feature-at-a-time summary this is the pairwise counterpart of.

Examples

```
feats <- c("mpg", "disp", "hp", "wt")

## All three coefficients on the same pairs
res <- summarize_association_stats(mtcars, feats)
round(res$pearson$corr, 3)
round(res$pearson$adj_pvalue, 4)

## A linear coefficient and a monotonic one on the same pair
c(pearson  = res$pearson$corr["mpg", "disp"],
  spearman = res$spearman$corr["mpg", "disp"])

## One method only
rho <- summarize_association_stats(mtcars, feats, methods = "spearman")
round(rho$spearman$corr, 3)
```

summarize_descriptive_stats *Descriptive summary of several features, optionally split by group*

Description

Reduces every feature to one row of sample size, central tendency, dispersion, quartiles, outlier fences and distribution shape. With a grouping vector the same row is produced per group level, so the summary lines up with the tests and plots that compare those levels.

Usage

```
summarize_descriptive_stats(data, feats, group = NULL, group_lv = NULL)
```

Arguments

data	A data.frame (or matrix) in wide format, one row per observation and one column per feature.
feats	Character vector of numeric column names in data to summarise, in output order.
group	Optional grouping vector with one entry per row of data. When NULL, all rows are summarised together and no group column is returned.
group_lv	Group levels to report, in output order. When NULL, the levels present in group are used: the factor levels if group is a factor, the sorted unique values otherwise. Rows belonging to any other level are dropped.

Details

Missing and non-finite values are dropped per feature and per group before anything is computed, so one Inf cannot turn a whole row into Inf or NaN; n_missing records how many were left out. A feature with no finite value in a group gives an all-NA row rather than aborting the summary.

skewness and excess_kurtosis are the bias-corrected G1 and G2 estimators used by SAS and SPSS, the same quantities as e1071::skewness(type = 2) and e1071::kurtosis(type = 2). They need three and four observations respectively, and a non-zero spread, and are NA otherwise.

cv is a ratio, so it only reads as relative dispersion when the values are positive. On data that crosses zero the mean shrinks towards it and the ratio explodes without the spread having changed.

out_lower_bound and out_upper_bound are the same fences that draw_grouped_boxplot() returns as lower_bound and upper_bound in box_summary_stats. They are where the whiskers may reach, not where they actually end.

Value

A data.frame with one row per feature, or per feature and group level when group is supplied. The leading columns are features and, when grouped, group. The remaining columns are:

n, n_missing Finite values the row is based on, and the values left out for being NA, NaN or infinite.

`mean`, `sd`, `var`, `se`, `cv` Mean, standard deviation, variance, standard error `sd / sqrt(n)` and coefficient of variation `sd / mean`.

`min`, `q1`, `median`, `q3`, `max`, `iqr` Five-number summary and the interquartile range. Quartiles come from `stats::quantile()` with its default type 7.

`out_lower_bound`, `out_upper_bound` Tukey outlier fences `q1 - 1.5 * iqr` and `q3 + 1.5 * iqr`.

`mad` Median absolute deviation from `stats::mad()`, scaled by the default 1.4826 so that it estimates `sd` for normal data.

`skewness`, `excess_kurtosis` Shape of the distribution, zero for a normal sample.

References

Joanes, D. N. and Gill, C. A. (1998). Comparing measures of sample skewness and kurtosis. *Journal of the Royal Statistical Society: Series D*, 47(1), 183-189.

See Also

`draw_grouped_boxplot()` to see the same quantities as a plot, and `compare_two_groups()` to test the difference between two levels.

Examples

```
feats <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")

## All rows together
summarize_descriptive_stats(iris, feats)

## One row per feature and species
by_species <- summarize_descriptive_stats(iris, feats, iris$Species)
by_species[by_species$features == "Petal.Length", ]

## Two of the three species, in a chosen order
summarize_descriptive_stats(iris, "Petal.Length", iris$Species,
                            c("virginica", "versicolor"))
```

Index