Package 'proDA'

December 2, 2025

Type Package

Title Differential Abundance Analysis of Label-Free Mass Spectrometry Data

Version 1.24.0

Description Account for missing values in label-free mass spectrometry data without imputation. The package implements a probabilistic dropout model that ensures that the information from observed and missing values are properly combined. It adds empirical Bayesian priors to increase power to detect differentially abundant proteins.

License GPL-3

Encoding UTF-8

LazyData false

RoxygenNote 7.1.0

Suggests testthat (>= 2.1.0), MSnbase, dplyr, stringr, readr, tidyr, tibble, limma, DEP, numDeriv, pheatmap, knitr, rmarkdown, BiocStyle

Imports stats, utils, methods, BiocGenerics, SummarizedExperiment, S4Vectors, extraDistr

URL https://github.com/const-ae/proDA

BugReports https://github.com/const-ae/proDA/issues

biocViews Proteomics, MassSpectrometry, DifferentialExpression, Bayesian, Regression, Software, Normalization, QualityControl

VignetteBuilder knitr

git_url https://git.bioconductor.org/packages/proDA

git_branch RELEASE_3_22

git_last_commit cf3d455

git_last_commit_date 2025-10-29

Repository Bioconductor 3.22

Date/Publication 2025-12-01

Author Constantin Ahlmann-Eltze [aut, cre] (ORCID: <https://orcid.org/0000-0002-3762-068X>), Simon Anders [ths] (ORCID: <https://orcid.org/0000-0003-4868-1805>)

Maintainer Constantin Ahlmann-Eltze <artjom31415@googlemail.com>

Contents

`.DollarNames.proDAFit` *Fluent use of accessor methods*

Description

The 'proDAFit' object overwrites the dollar function to make it easy to call functions to access values inside the object. This has the advantage that it is very easy to discover the relevant methods but nonetheless have an isolated implementation. Unlike the `` `@` `` operator which directly accesses the underlying implementation, the `` `$` `` operator only exposes a limited set of functions

- abundances
- hyper_parameters
- feature_parameters
- coefficients
- convergence
- design
- reference_level

- result_names
- coefficient_variance_matrices
- colData
- rowData

Usage

```
## S3 method for class 'proDAFit'
.DollarNames(x, pattern = "")

## S4 method for signature 'proDAFit'
x$name

## S4 replacement method for signature 'proDAFit'
x$name <- value
```

Arguments

x	an object of class 'proDAFit' produced by `proDA()`
pattern	the regex pattern that is provided by the IDE
name	one of the functions listed above
value	**Warning:** modifying the content of a 'proDAFit' object is not allowed

Value

whatever the function called `name` returns.

See Also

accessor_methods for more documentation on the accessor functions.

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)

# The two styles are identical
design(fit)
fit$design

# More functions
fit$abundances
```

abundances *Get the abundance matrix*

Description

Get the abundance matrix

Usage

```
abundances(object, ...)
```

Arguments

object	the object to get from
...	additional arguments used by the concrete implementation

Value

the original matrix that was fitted

See Also

accessor_methods for the implementation for a 'proDAFit' object

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
abundances(fit)
```

accessor_methods *Get different features and elements of the 'proDAFit' object*

Description

The functions listed below can all be accessed using the fluent dollar notation (ie. `fit$abundances[1:3,1:3]`) without any additional parentheses.

Usage

```
## S4 method for signature 'proDAFit'
abundances(object)

## S4 method for signature 'proDAFit'
design(object, formula = FALSE)

## S4 method for signature 'proDAFit'
hyper_parameters(object)
```

```
## S4 method for signature 'proDAFit'
feature_parameters(object)

## S4 method for signature 'proDAFit'
coefficients(object)

## S4 method for signature 'proDAFit'
coefficient_variance_matrices(object)

## S4 method for signature 'proDAFit'
reference_level(object)

## S4 method for signature 'proDAFit'
convergence(object)
```

Arguments

`object`	the 'proDAFit' object
`formula`	specific argument for the `design` function to get the formula that was used to create the linear model. If no formula was used `NULL` is returned.

Value

See the documentation of the generics to find out what each method returns

---

`as_replicate` *Get numeric vector with the count of the replicate for each element*

Description

For a vector with repeated values return a vector where each element is the count how often the element was observed previously

Usage

```
as_replicate(x)
```

Arguments

`x`	a vector with repeated elements

Value

numeric vector

See Also

`order`, `rank`

Examples

```
x <- c("a", "b", "a", "b", "b", "d")
all(proDA:::as_replicate(x) == c(1,1,2,2,3,1))
```

`coefficients` *Get the coefficients*

Description

Get the coefficients

Usage

```
coefficients(object, ...)
```

Arguments

`object`	the object to get from
`...`	additional arguments used by the concrete implementation

Value

a numeric matrix of size ‘nrow(fit) * p‘

See Also

accessor_methods for the implementation for a ’proDAFit’ object

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
coefficients(fit)
```

`coefficient_variance_matrices` *Get the coefficients*

Description

Get the coefficients

Usage

```
coefficient_variance_matrices(object, ...)
```

Arguments

`object`	the object to get from
`...`	additional arguments used by the concrete implementation

Value

a list with as many entries as rows in the data. Each element is a p*p matrix

See Also

accessor_methods for the implementation for a 'proDAFit' object

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
coefficient_variance_matrices(fit)
```

---

convergence *Get the convergence information*

Description

Get the convergence information

Usage

```
convergence(object, ...)
```

Arguments

object	the object to get from
...	additional arguments used by the concrete implementation

Value

a list with information on the convergence

See Also

accessor_methods for the implementation for a 'proDAFit' object

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
convergence(fit)
```

distance_sq *Square distance between two Gaussian distributions*

Description

The function takes the mean and the diagonal of the covariance matrix as vector and calculates the mean and variance of their distance distribution. The formulas are based on [1] page 53.

Usage

```
distance_sq(mu1, sigma1, mu2, sigma2)
```

Value

a list with elements ‘mean‘ and ‘var‘

1. Mathai, A. & Provost, S. Quadratic Forms in Random Variables. (1992).

dist_approx *Calculate an approximate distance for 'object'*

Description

Calculate an approximate distance for 'object'

Usage

```
dist_approx(object, ...)
```

Arguments

object	the object for which the distance is approximated
...	additional arguments used by the concrete implementation

Value

a list with two elements: ‘mean‘ and ‘sd‘ both are formally of class "dist"

See Also

dist for the base R function and dist_approx() concrete implementation for 'proDAFit' objects

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
dist_approx(fit)
```

`dist_approx_impl` *Distance method for 'proDAFit' object*

Description

The method calculates either the euclidean distance between samples or proteins taking into account the missing values and the associated uncertainty. Because with missing value no single deterministic distance can be calculated two objects are returned: the mean and the associated standard deviation of the distance estimates.

Usage

```
## S4 method for signature 'proDAFit'
dist_approx(object, by_sample = TRUE, blind = TRUE)

## S4 method for signature 'SummarizedExperiment'
dist_approx(object, by_sample = TRUE, blind = TRUE, ...)

## S4 method for signature 'ANY'
dist_approx(object, by_sample = TRUE, blind = TRUE, ...)
```

Arguments

`object`	the 'proDAFit' object for which we calculate the distance or a matrix like object for which 'proDAFit' is created internally
`by_sample`	a boolean that indicates if the distances is calculated between the samples ('by_sample = TRUE') or between the proteins ('by_sample = FALSE'). Default: 'TRUE'
`blind`	fit an intercept model for the missing values to make sure that the results are not biased for the expected result. Default: 'TRUE'
`...`	additional arguments to `proDA()` in case object is a `SummarizedExperiment` or a `matrix`

Value

a list with two elements: 'mean' and 'sd' both are formally of class "dist"

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
dist_approx(fit)
```

feature_parameters *Get the feature parameters*

Description

Get the feature parameters

Usage

```
feature_parameters(object, ...)
```

Arguments

object	the object to get from
...	additional arguments used by the concrete implementation

Value

a data.frame with information on each fit

See Also

accessor_methods for the implementation for a 'proDAFit' object

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
feature_parameters(fit)
```

generate_synthetic_data *Generate a dataset according to the probabilistic dropout model*

Description

Generate a dataset according to the probabilistic dropout model

Usage

```
generate_synthetic_data(
  n_proteins,
  n_conditions = 2,
  n_replicates = 3,
  frac_changed = 0.1,
  dropout_curve_position = 18.5,
  dropout_curve_scale = -1.2,
  location_prior_mean = 20,
  location_prior_scale = 4,
```

```
  variance_prior_scale = 0.05,
  variance_prior_df = 2,
  effect_size = 2,
  return_summarized_experiment = FALSE
)
```

Arguments

`n_proteins`	the number of rows in the dataset
`n_conditions`	the number of conditions. Default: 2
`n_replicates`	the number of replicates per condition. Can either be a single number or a vector with `length(n_replicates) == n_conditions`. Default: 3
`frac_changed`	the fraction of proteins that actually differ between the conditions. Default: 0.1
`dropout_curve_position`	the point where the chance to observe a value is 50%. Can be a single number or a vector of `length(dropout_curve_position) == n_conditions * n_replicates`. Default: 18.5
`dropout_curve_scale`	The width of the dropout curve. Negative numbers mean that lower intensities are more likely to be missing. Can be a single number or a vector of `length(dropout_curve_position) == n_conditions * n_replicates`. Default: -1.2
`location_prior_mean`, `location_prior_scale`	the position and the variance around which the individual condition means (`t_mu`) scatter. Default: 20 and 4
`variance_prior_scale`, `variance_prior_df`	the scale and the degrees of freedom of the inverse Chi-squared distribution used as a prior for the variances. Default: 0.05 and 2
`effect_size`	the standard deviation that is used to draw different values for the `frac_changed` part of the proteins. Default: 2
`return_summarized_experiment`	a boolean indicator if the method should return a `SummarizedExperiment` object instead of a list. Default: `FALSE`

Value

a list with the following elements

Y the intensity matrix including the missing values

Z the intensity matrix before dropping out values

t_mu a matrix with `n_proteins` rows and `n_conditions` columns that contains the underlying means for each protein

t_sigma2 a vector with the true variances for each protein

changed a vector with boolean values if the protein is actually changed

group the group structure mapping samples to conditions

if `return_summarized_experiment` is `FALSE`. Otherwise returns a `SummarizedExperiment` with the same information.

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
names(syn_data)
head(syn_data$Y)

# Returning a SummarizedExperiment
se <- generate_synthetic_data(n_proteins = 10, return_summarized_experiment = TRUE)
se
head(SummarizedExperiment::assay(se))
```

`hyper_parameters` *Get the hyper parameters*

Description

Get the hyper parameters

Usage

```
hyper_parameters(object, ...)
```

Arguments

`object`	the object to get from
`...`	additional arguments used by the concrete implementation

Value

a list with the values for each fitted hyper-parameter

See Also

accessor_methods for the implementation for a 'proDAFit' object

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
hyper_parameters(fit)
```

invprobit *Inverse probit function*

Description

Calculate the values of the sigmoidal function that is defined by the cumulative normal distribution function (pnorm). This method provides a convenient wrapper for the pnorm that automatically handles negative zeta and is more consistent in its naming.

Usage

```
invprobit(x, rho, zeta, log = FALSE, oneminus = FALSE)
```

Arguments

x	numeric vector
rho	numeric vector of length 1 or the same length as x. Specifies the inflection point of the inverse probit curve.
zeta	numeric vector of length 1 or the same length as x. Specifies the scale of the curve at the inflection point of the inverse probit curve.
log	boolean if the log of the result is returned
oneminus	boolean if one minus the result is returned

Value

a numeric vector of length(x).

Examples

```
xg <- seq(-5, 5, length.out=101)
plot(xg, invprobit(xg, rho=-2, zeta=-0.3))
```

invprobit_fast *Same thing as invprobit, but without the parameter validation*

Description

Same thing as invprobit, but without the parameter validation

Usage

```
invprobit_fast(x, rho, zeta, log = FALSE, oneminus = FALSE)
```

Value

a numeric vector of length(x)

median_normalization *Column wise median normalization of the data matrix*

Description

The method calculates for each sample the median change (i.e. the difference between the observed value and the row average) and subtracts it from each row. Missing values are ignored in the procedure. The method is based on the assumption that a majority of the rows did not change.

Usage

```
median_normalization(X, spike_in_rows = NULL)
```

Arguments

X	a matrix or SummarizedExperiment of proteins and samples
spike_in_rows	a numeric or boolean vector that is used to to normalize the intensities across samples. Default: NULL which means that all rows are used.

Value

the normalized matrix

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
normalized_data <- median_normalization(syn_data$Y)
normalized_data

# If we assume that the first 5 proteins are spike-ins
normalized_data2 <- median_normalization(syn_data$Y, spike_in_rows = 1:5)
```

mply_dbl *apply function that always returns a numeric matrix*

Description

The function is modeled after 'vapply', but always returns a matrix with one row for each iteration. You need to provide the number of elements each function call produces beforehand (i.e. the number of resulting columns). For a more flexible version where you don't need to provide the number of columns see msply_dbl

Usage

```
mply_dbl(x, FUN, ncol = 1, ...)

msply_dbl(x, FUN, ...)
```

Arguments

x	a vector that will be passed to ‘vapply‘ or a matrix that will be passed to apply with MARGIN=1.
FUN	the function that returns a vector of length ncol
ncol	the length of the vector returned by ‘FUN‘.
...	additional arguments to FUN

Value

a matrix of size length(x) x ncol

Functions

- mply_dbl: apply function that always returns a numeric matrix
- msply_dbl: flexible version that automatically infers the number of columns

Examples

```
# Behaves similar to sapply(), but it always returns a matrix
t(sapply(1:5, function(i) c(i - i/3, i, i + i/3)))
proDA:::mply_dbl(1:5, function(i) c(i - i/3, i, i + i/3), ncol=3)

# Which can avoid some bad surprises
t(sapply(1:5, identity))
proDA:::mply_dbl(1:5, identity)


# Works also with matrix input
mat <- matrix(1:20, ncol=4)
mat
proDA:::msply_dbl(mat, function(i) rep(i, each=2))
```

---

pd_lm *Fit a single linear probabilistic dropout model*

Description

The function works similar to the classical lm but with special handling of NA's. Whereas lm usually just ignores response value that are missing, pd_lm applies a probabilistic dropout model, that assumes that missing values occur because of the dropout curve. The dropout curve describes for each position the chance that that a value is missed. A negative dropout_curve_scale means that the lower the intensity was, the more likely it is to miss the value.

Usage

```
pd_lm(
  formula,
  data = NULL,
  subset = NULL,
  dropout_curve_position,
  dropout_curve_scale,
  location_prior_mean = NULL,
  location_prior_scale = NULL,
  variance_prior_scale = NULL,
  variance_prior_df = NULL,
  location_prior_df = 3,
  method = c("analytic_hessian", "analytic_grad", "numeric"),
  verbose = FALSE
)
```

Arguments

`formula`	a formula that specifies a linear model
`data`	an optional data.frame whose columns can be used to specify the `formula`
`subset`	an optional selection vector for data to subset it
`dropout_curve_position`	the value where the chance to observe a value is 50%. Can either be a single value that is repeated for each row or a vector with one element for each row. Not optional.
`dropout_curve_scale`	the width of the dropout curve. Smaller values mean that the sigmoidal curve is steeper. Can either be a single value that is repeated for each row or a vector with one element for each row. Not optional.
`location_prior_mean`, `location_prior_scale`	the optional mean and variance of the prior around which the predictions are supposed to scatter. If no value is provided no location regularization is applied.
`variance_prior_scale`, `variance_prior_df`	the optional scale and degrees of freedom of the variance prior. If no value is provided no variance regularization is applied.
`location_prior_df`	The degrees of freedom for the t-distribution of the location prior. If it is large (> 30) the prior is approximately Normal. Default: 3
`method`	one of 'analytic_hessian', 'analytic_gradient', or 'numeric'. If 'analytic_hessian' the `nlminb` optimization routine is used, with the hand derived first and second derivative. Otherwise, `optim` either with or without the first derivative is used.
`verbose`	boolean that signals if the method prints informative messages. Default: `FALSE`.

Value

a list with the following entries

coefficients a named vector with the fitted values

coef_variance_matrix a p*p matrix with the variance associated with each coefficient estimate

n_approx the estimated "size" of the data set (n_hat - variance_prior_df)

df the estimated degrees of freedom (n_hat - p)

s2 the estimated unbiased variance

n_obs the number of response values that were not 'NA'

Examples

```
# Without missing values
y <- rnorm(5, mean=20)
lm(y ~ 1)
pd_lm(y ~ 1,
      dropout_curve_position = NA,
      dropout_curve_scale = NA)

# With some missing values
y <- c(23, 21.4, NA)
lm(y ~ 1)
pd_lm(y ~ 1,
      dropout_curve_position = 19,
      dropout_curve_scale = -1)


# With only missing values
y <- c(NA, NA, NA)
# lm(y ~ 1)  # Fails
pd_lm(y ~ 1,
      dropout_curve_position = 19,
      dropout_curve_scale = -1,
      location_prior_mean = 21,
      location_prior_scale = 3,
      variance_prior_scale = 0.1,
      variance_prior_df = 2)
```

---

pd_lm.fit *The work horse for fitting the probabilistic dropout model*

Description

If there is no location and variance moderation and no missing values, the model is fitted with 'lm'.

Usage

```
pd_lm.fit(
  y,
  X,
  dropout_curve_position,
  dropout_curve_scale,
  location_prior_mean = NULL,
  location_prior_scale = NULL,
  variance_prior_scale = NULL,
  variance_prior_df = NULL,
  location_prior_df = 3,
```

```
  method = c("analytic_hessian", "analytic_grad", "numeric"),
  verbose = FALSE
)
```

Value

a list with the following entries

coefficients a named vector with the fitted values

n_approx the estimated "size" of the data set (n_hat - variance_prior_df)

df the estimated degrees of freedom (n_hat - p)

s2 the estimated unbiased variance

n_obs the number of response values that were not 'NA'

---

pd_row_t_test *Row-wise tests of difference using the probabilistic dropout model*

Description

This is a helper function that combines the call of `proDA()` and `test_diff()`. If you need more flexibility use those functions.

Usage

```
pd_row_t_test(
  X,
  Y,
  moderate_location = TRUE,
  moderate_variance = TRUE,
  alternative = c("two.sided", "greater", "less"),
  pval_adjust_method = "BH",
  location_prior_df = 3,
  max_iter = 20,
  epsilon = 0.001,
  return_fit = FALSE,
  verbose = FALSE
)

pd_row_f_test(
  X,
  ...,
  groups = NULL,
  moderate_location = TRUE,
  moderate_variance = TRUE,
  pval_adjust_method = "BH",
  location_prior_df = 3,
  max_iter = 20,
  epsilon = 0.001,
  return_fit = FALSE,
  verbose = FALSE
)
```

Arguments

X, Y, ...	the matrices for condition 1, 2 and so on. They must have the same number of rows.
moderate_location	boolean values to indicate if the location and the variances are moderated. Default: TRUE
moderate_variance	boolean values to indicate if the location and the variances are moderated. Default: TRUE
alternative	a string that decides how the hypothesis test is done. This parameter is only relevant for the Wald-test specified using the 'contrast' argument. Default: "two.sided"
pval_adjust_method	a string the indicates the method that is used to adjust the p-value for the multiple testing. It must match the options in p.adjust. Default: "BH"
location_prior_df	the number of degrees of freedom used for the location prior. A large number (> 30) means that the prior is approximately Normal. Default: 3
max_iter	the maximum of iterations proDA() tries to converge to the hyper-parameter estimates. Default: 20
epsilon	if the remaining error is smaller than epsilon the model has converged. Default: 1e-3
return_fit	boolean that signals that in addition to the data.frame with the hypothesis test results, the fit from proDA() is returned. Default: FALSE
verbose	boolean that signals if the method prints messages during the fitting. Default: FALSE
groups	a factor or character vector with that assignes the columns of X to different conditions. This parameter is only applicable for the F-test and must be specified if only a single matrix is provided.

Details

The pd_row_t_test is not actually doing a t-test, but rather a Wald test. But, as the two are closely related and term t-test is more widely understood, we choose to use that name.

Value

If return_fit == FALSE a data.frame is returned with the content that is described in test_diff.

If return_fit == TRUE a list is returned with two elements: fit with a reference to the object returned from proDA() and a test_result() with the data.frame returned from test_diff().

See Also

proDA and test_diff for more flexible versions. The function was inspired by the rowFtests function in the genefilter package.

Examples

```
data1 <- matrix(rnorm(10 * 3), nrow=10)
data2 <- matrix(rnorm(10 * 4), nrow=10)
data3 <- matrix(rnorm(10 * 2), nrow=10)

# Comparing two datasets
pd_row_t_test(data1, data2)

# Comparing multiple datasets
pd_row_f_test(data1, data2, data3)

# Alternative
data_comb <- cbind(data1, data2, data3)
pd_row_f_test(data_comb,
   groups = c(rep("A",3), rep("B", 4), rep("C", 2)))

# t.test, lm, pd_row_t_test, and pd_row_f_test are
# approximately equivalent on fully observed data
set.seed(1)
x <- rnorm(5)
y <- rnorm(5, mean=0.3)

t.test(x, y)
summary(lm(c(x, y) ~ cond,
           data = data.frame(cond = c(rep("x", 5),
                                      rep("y", 5)))))$coefficients[2,]
pd_row_t_test(matrix(x, nrow=1), matrix(y, nrow=1),
              moderate_location = FALSE,
              moderate_variance = FALSE)
pd_row_f_test(matrix(x, nrow=1), matrix(y, nrow=1),
              moderate_location = FALSE,
              moderate_variance = FALSE)
```

`predict,proDAFit-method`

Predict the parameters or values of additional proteins

Description

This function can either predict the abundance matrix for proteins (`type = "response"`) without missing values according to the linear probabilistic dropout model, fitted with `proDA()`. Or, it can predict the feature parameters for additional proteins given their abundances including missing values after estimating the hyper-parameters on a dataset with the same sample structure (`type = "feature_parameters"`).

Usage

```
## S4 method for signature 'proDAFit'
predict(
  object,
  newdata,
```

```
  newdesign,
  type = c("response", "feature_parameters"),
  ...
)
```

Arguments

`object`	an 'proDAFit' object that is produced by `proDA()`.
`newdata`	a matrix or a SummarizedExperiment which contains the new abundances for which values are predicted.
`newdesign`	a formula or design matrix that specifies the new structure that will be fitted
`type`	either "response" or "feature_parameters". Default: `"response"`
`...`	additional parameters for the construction of the 'proDAFit' object.

Details

Note: this method behaves a little different from what one might expect from the classical `predict.lm()` function, because `object` is not just a single set of coefficients for one fit, but many fits (ie. one for each protein) with some more hyper-parameters. The classical `predict` function predicts the response for new samples. This function does not support this, instead it is useful for getting a matrix without missing values for additional proteins.

Value

If `type = "response"` a matrix with the same dimensions as `object`. Or, if `type = "feature_parameters"` a 'proDAFit' object with the same hyper-parameters and column data as `object`, but new fitted `rowData()`.

---

proDA *Main function to fit the probabilistic dropout model*

Description

The function fits a linear probabilistic dropout model and infers the hyper-parameters for the location prior, the variance prior, and the dropout curves. In addition it infers for each protein the coefficients that best explain the observed data and the associated uncertainty.

Usage

```
proDA(
  data,
  design = ~1,
  col_data = NULL,
  reference_level = NULL,
  data_is_log_transformed = TRUE,
  moderate_location = TRUE,
  moderate_variance = TRUE,
  location_prior_df = 3,
  n_subsample = nrow(data),
  max_iter = 20,
```

```
  epsilon = 0.001,
  verbose = FALSE,
  ...
)
```

Arguments

data	a matrix like object (`matrix()`, `SummarizedExperiment()`, or anything that can be cast to `SummarizedExperiment()` (eg. 'MSnSet', 'eSet', ...)) with one column per sample and one row per protein. Missing values should be coded as `NA`.
design	a specification of the experimental design that is used to fit the linear model. It can be a `model.matrix()` with one row for each sample and one column for each coefficient. It can also be a formula with the entries referring to global objects, columns in the `col_data` argument or columns in the `colData(data)` if data is a `SummarizedExperiment`. Thirdly, it can be a vector that for each sample specifies the condition of that sample. Default: `~ 1`, which means that all samples are treated as if they are in the same condition.
col_data	a data.frame with one row for each sample in `data`. Default: `NULL`
reference_level	a string that specifies which level in a factor coefficient is used for the intercept. Default: `NULL`
data_is_log_transformed	the raw intensities from mass spectrometry experiments have a linear mean-variance relation. This is undesirable and can be removed by working on the log scale. The easiest way to find out if the data is already log- transformed is to see if the intensities are in the range of '0' to '100' in which case they are transformed or if they rather are between '1e5' to '1e12', in which case they are not. Default: `TRUE`
moderate_location, moderate_variance	boolean values to indicate if the location and the variances are moderated. Default: `TRUE`
location_prior_df	the number of degrees of freedom used for the location prior. A large number (> 30) means that the prior is approximately Normal. Default: `3`
n_subsample	the number of proteins that are used to estimate the hyper-parameter. Reducing this number can speed up the fitting, but also mean that the final estimate is less precise. By default all proteins are used. Default: `nrow(data)`
max_iter	the maximum of iterations `proDA()` tries to converge to the hyper-parameter estimates. Default: `20`
epsilon	if the remaining error is smaller than `epsilon` the model has converged. Default: `1e-3`
verbose	boolean that signals if the method prints messages during the fitting. Default: `FALSE`
...	additional parameters for the construction of the 'proDAFit' object

Details

By default, the method is moderating the locations and the variance of each protein estimate. The variance moderation is fairly standard in high-throughput experiments and can boost the power to

detect differentially abundant proteins. The location moderation is important to handle the edge case where in one condition a protein is not observed in any sample. In addition it can help to get more precise estimates of the difference between conditions. Unlike 'DESeq2', which moderates the coefficient estimates (ie. the "betas") to be centered around zero, 'proDA' penalizes predicted intensities that strain far from the other observed intensities.

Value

An object of class 'proDAFit'. The object contains information on the hyper-parameters and feature parameters, the convergence, the experimental design etc. Internally, it is a sub-class of `SummarizedExperiment` which means the object is subsettable. The '$'-operator is overloaded for this object to make it easy to discover applicable functions.

Examples

```
# Quick start

# Import the proDA package if you haven't already done so
# library(proDA)
set.seed(1)
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
fit
result_names(fit)
test_diff(fit, Condition_1 - Condition_2)

# SummarizedExperiment
se <- generate_synthetic_data(n_proteins = 10,
                      return_summarized_experiment = TRUE)
se
proDA(se, design = ~ group)

# Design using model.matrix()
data_mat <- matrix(rnorm(5 * 10), nrow=10)
colnames(data_mat) <- paste0("sample", 1:5)
annotation_df <- data.frame(names = paste0("sample", 1:5),
                      condition = c("A", "A", "A", "B", "B"),
                      age = rnorm(5, mean=40, sd=10))

design_mat <- model.matrix(~ condition + age,
                           data=annotation_df)
design_mat
proDA(data_mat, design_mat, col_data = annotation_df)
```

`proDAFit-class` *proDA Class Definition*

Description

proDA Class Definition

proDA_package *proDA: Identify differentially abundant proteins in label-free mass spectrometry*

Description

Account for missing values in label-free mass spectrometry data without imputation. The package implements a probabilistic dropout model that ensures that the information from observed and missing values are properly combined. It adds empirical Bayesian priors to increase power to detect differentially abundant proteins.

reference_level *Get the reference level*

Description

Get the reference level

Usage

```
reference_level(object, ...)
```

Arguments

object	the object to get from
...	additional arguments used by the concrete implementation

Value

a string

See Also

accessor_methods for the implementation for a 'proDAFit' object

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups, reference_level = "Condition_1")
reference_level(fit)
```

result_names *Get the result_names*

Description

Get the result_names

Usage

```
result_names(fit, ...)
```

Arguments

fit	the fit to get the result_names from
...	additional arguments used by the concrete implementation

Value

a character vector

Examples

```
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
result_names(fit)
```

test_diff *Identify differentially abundant proteins*

Description

The 'test_diff()' function is used to test coefficients of a 'proDAFit' object. It provides a Wald test to test individual coefficients and a likelihood ratio F-test to compare the original model with a reduced model. The result_names method provides a quick overview which coefficients are available for testing.

Usage

```
test_diff(
  fit,
  contrast,
  reduced_model = ~1,
  alternative = c("two.sided", "greater", "less"),
  pval_adjust_method = "BH",
  sort_by = NULL,
  decreasing = FALSE,
  n_max = Inf,
  verbose = FALSE
)
```

```
## S4 method for signature 'proDAFit'
result_names(fit)
```

Arguments

fit	an object of class 'proDAFit'. Usually, this is produced by calling `proDA()`
contrast	an expression or a string specifying which contrast is tested. It can be a single coefficient (to see the available options use `result_names(fit)`) or any linear combination of them. The contrast is always compared against zero. Thus, to find out if two coefficients differ use `coef1 - coef2`. Remember if the coefficient is not a valid identifier in R, to escape it using back ticks. For example if you test the interaction of A and B use `` `A:B` ``.
reduced_model	If you don't want to test an individual coefficient, you can can specify a reduced model and compare it with the original model using an F-test. This is useful to find out how a set of parameters affect the goodness of the fit. If neither a `contrast`, nor a `reduced_model` is specified, by default a comparison with an intercept model (ie. just the average across conditions) is done. Default: `~ 1`.
alternative	a string that decides how the hypothesis test is done. This parameter is only relevant for the Wald-test specified using the 'contrast' argument. Default: `"two.sided"`
pval_adjust_method	a string the indicates the method that is used to adjust the p-value for the multiple testing. It must match the options in `p.adjust`. Default: `"BH"`
sort_by	a string that specifies the column that is used to sort the resulting data.frame. Default: `NULL` which means the result is sorted by the order of the input matrix.
decreasing	a boolean to indicate if the order is reversed. Default: `FALSE`
n_max	the maximum number of rows returned by the method. Default: `Inf`
verbose	boolean that signals if the method prints informative messages. Default: `FALSE`.

Details

To test if coefficient is different from zero with a Wald test use the `contrast` function argument. To test if two models differ with an F-test use the `reduced_model` argument. Depending on the test that is conducted, the functions returns slightly different data.frames.

The function is designed to follow the principles of the base R test functions (ie. `t.test` and `wilcox.test`) and the functions designed for collecting the results of high-throughput testing (ie. `limma::topTable` and `DESeq2::results`).

Value

The 'result_names()' function returns a character vector.

The 'test_diff()' function returns a `data.frame` with one row per protein with the key parameters of the statistical test. Depending what kind of test (Wald or F test) the content of the 'data.frame' differs.

The Wald test, which can considered equivalent to a t-test, returns a 'data.frame' with the following columns:

name the name of the protein, extracted from the rowname of the input matrix

pval the p-value of the statistical test

adj_pval the multiple testing adjusted p-value

diff the difference that particular coefficient makes. In differential expression analysis this value is also called log fold change, which is equivalent to the difference on the log scale.

t_statistic the `diff` divided by the standard error `se`

se the standard error associated with the `diff`

df the degrees of freedom, which describe the amount of available information for estimating the `se`. They are the sum of the number of samples the protein was observed in, the amount of information contained in the missing values, and the degrees of freedom of the variance prior.

avg_abundance the estimate of the average abundance of the protein across all samples.

n_approx the approximated information available for estimating the protein features, expressed as multiple of the information contained in one observed value.

n_obs the number of samples a protein was observed in

The F-test returns a 'data.frame' with the following columns

name the name of the protein, extracted from the rowname of the input matrix

pval the p-value of the statistical test

adj_pval the multiple testing adjusted p-value

f_statistic the ratio of difference of normalized deviances from original model and the reduced model, divided by the standard deviation.

df1 the difference of the number of coefficients in the original model and the number of coefficients in the reduced model

df2 the degrees of freedom, which describe the amount of available information for estimating the `se`. They are the sum of the number of samples the protein was observed in, the amount of information contained in the missing values, and the degrees of freedom of the variance prior.

avg_abundance the estimate of the average abundance of the protein across all samples.

n_approx the information available for estimating the protein features, expressed as multiple of the information contained in one observed value.

n_obs the number of samples a protein was observed in

See Also

The contrast argument is inspired by `limma::makeContrasts`.

Examples

```
# "t-test"
syn_data <- generate_synthetic_data(n_proteins = 10)
fit <- proDA(syn_data$Y, design = syn_data$groups)
result_names(fit)
test_diff(fit, Condition_1 - Condition_2)


suppressPackageStartupMessages(library(SummarizedExperiment))
se <- generate_synthetic_data(n_proteins = 10,
                              n_conditions = 3,
                              return_summarized_experiment = TRUE)
colData(se)$age <- rnorm(9, mean=45, sd=5)
colData(se)
fit <- proDA(se, design = ~ group + age)
```

```
result_names(fit)
test_diff(fit, "groupCondition_2",
          n_max = 3, sort_by = "pval")

# F-test
test_diff(fit, reduced_model = ~ group)
```

`%zero_dom_mat_mult%` *Helper function that makes sure that NA * 0 = 0 in matrix multiply*

Description

Helper function that makes sure that NA * 0 = 0 in matrix multiply

Usage

```
X %zero_dom_mat_mult% Y
```

Arguments

`X`	a matrix of size ‘n*m‘
`Y`	a matrix of size ‘m*p‘

Value

a matrix of size ‘n*p‘

Index