KdeggeR User Manual

Barbora Salovska

Last edited Tuesday 26 November 2024

Contents

1 Load the package

1.1 Install dependencies

```r
# Required packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(dplyr, outliers, purrr, stringr, tibble)

# Optional R package for robust linear model fitting
install.packages("MASS")
```

1.2 Install KdeggeR

```r
library("devtools")
install_github("yslproteomics/KdeggeR", build_vignettes = TRUE)
```

1.3 Open the vignette

```r
vignette("KdeggerUserManual", package = "KdeggeR")
```

1.4 Load KdeggeR

```r
library(KdeggeR)
```

2 Required files

1. Precursor-level data exported in .tsv format
2. Design table saved in .tsv format
3. OPTIONAL: Table with experimentally determined kcd values in .tsv format

2.1 Data

- The data can be either provided as a path to the file or can be a data.frame already loaded in the R environment.
- The data must contain light and heavy intensity columns, a protein id column, and a precursor id column. Other columns are optional.
- If the data has been pre-loaded in the R environment, the data must have row names corresponding to precursor ids.
- See examples in the code chunks below.

2.1.1 Spectronaut data, labeled workflow

- Required columns: `PG.ProteinGroups`, `EG.PrecursorId`
- Light intensities: columns ending with `EG.Channel1Quantity`
- Heavy intensities: columns ending with `EG.Channel2Quantity`
- Example data are provided, see `KdeggeR::example_spectronaut` to check the expected data structure

```
KdeggeR::example_spectronaut %>%
  dplyr::glimpse()
```

2.1.2 Spectronaut data, ISW workflow

- In SN19, the same columns can be exported and use
- In older SN versions:
  - The light channel quantities are named as `EG.ReferenceQuantity`
  - The heavy channel quantities are named `EG.TargetQuantity`
  - In this case, rename the columns, so they match to the expected format, e.g., using the code in the chunk below.

```
KdeggeR::example_spectronaut_isw %>%
  dplyr::rename_with(~gsub("EG.TargetQuantity..Settings.", "EG.Channel2Quantity", .),
                     .cols = dplyr::ends_with("EG.TargetQuantity..Settings.")) %>%
  dplyr::rename_with(~gsub("EG.ReferenceQuantity..Settings.", "EG.Channel1Quantity", .),
                     .cols = dplyr::ends_with("EG.ReferenceQuantity..Settings."))
```

2.1.3 DIA-NN and Fragpipe data, plexDIA

- Recommended output:
- Required columns: `Protein.Group`, `Precursor.Id`
- Light intensities: columns ending with `.L`
- Heavy intensities: columns ending with `.H`
- Example data are provided, see `KdeggeR::example_diann` to check the expected data structure

```
KdeggeR::example_diann %>%
  dplyr::glimpse()
```

2.2 Design table

- The design can be either provided as a path to the file or can be a data.frame already loaded in the R environment.
- If the data has been pre-loaded in the R environment, the table must have row names corresponding to raw file names.
- Mandatory columns:
  1. sample
  2. time (numeric, in hours)
  3. color (enables data plotting in post-analysis)
- Optional columns:

1. replicates (numeric) - needed replicate design analysis. Note replicates of the same sample should have the same value in the column sample of the design table if replicate design is used. See examples.

2. condition

2.2.1 Design table without replicate design

- If there are no replicates to be averaged before data processing.
- Unique sample names must be specified in the `sample` column
- The labeling time point must be specified in the `time` column

```r
# Examine the data structure
KdeggeR::example_spectronaut_design %>%
  dplyr::glimpse()

# Examine the row names
row.names(KdeggeR::example_spectronaut_design)
```

2.2.2 Design table with replicate design

- If there are replicates that needs to be combined before analysis, the `sample` column needs to be specified accordingly to group the replicates
- The `replicate` column must contain information about the replicates (numeric).
- The labeling time point must be specified in the `time` column

Note, if replicate design is used, the data will be combined in the beginning of the processing, and all subsequent steps are performed with the averaged data (such as RIA calculation, data filtering, kloss modeling, etc..).

```r
# Examine the data structure
KdeggeR::example_spectronaut_design_replicates %>%
  dplyr::glimpse()

# Examine the row names
row.names(KdeggeR::example_spectronaut_design_replicates)
```

2.2.3 Generate design table based on the results file

- The `generate_design_template()` function can be used to generate a customizable design table template based on the data file.
- In this case, a path to the data file must be provided - example tsv files are provided in the data directory.
- A tsv file is exported, which can be customized by the user, saved as a .tsv file and used as an input for the `generatepSILACObject()` function.

```r
# use path to the file
input_data <- paste(.libPaths()[1],
                    "KdeggeR/data_tsv/example_data_pSILAC_SN19.tsv",
                    sep = "/")
```

```
KdeggeR::generateDesignTemplate(dataset = input_data, inputDataType = "spectronaut")

# use path to the file
input_data <- paste(.libPaths()[1],
                    "KdeggeR/data_tsv/example_data_pSILAC_diann_1,9.tsv",
                    sep = "/")

KdeggeR::generateDesignTemplate(dataset = input_data, inputDataType = "diann")
```

2.3 kcd table

- A simple kcd table with two columns can be provided to be used for kdeg calculation.
- The table must contain a `sample` column identical to the `sample` column in the design table, and a `kcd` column containing experimentally-derived cell doubling rates.
- See an example table below.

```
KdeggeR::example_kcd %>%
  glimpse()
```

3 Generate pSILAC object

3.1 Function description

The pSILAC object is generated using the design table and the data using the `generatepSILACobject()` function. The function prepares the R object, filters and/or averages the intensity data, and calculates the RIA(Light), ln(H/L + 1), and NLI values, which are then used for downstream processing.

The input data can be either loaded on R environment or provided as a path to the file. The `inputDataType` parameter must be correctly specified as `spectronaut`, `diann`, `fragpipe`, `maxquant`, or `openswath`.

For no replicate averaging, please set the `aggregate.replicates` parameter to `NA`. In this workflow, all unique conditions defined using the sample column in the design table will be processed independently. For replicate averaging, set the `aggregate.replicates` parameter to either `mean` or `median` and provide a design table with the replicate indicated (as described above). When the replicate design is activated, the light and heavy intensities will be averaged using the selected method before any downstream filtering and analysis.

Specify the number of cores that will be used for the analysis using the paramater `ncores`. By default set to 1, if NULL, the number of cores will be determined as `detectCores() - 1`.

The `filterPeptides` parameter will remove peptides with no lysine or arginine residues and can be used by default in a classic pSILAC experiment using stable isotopes of both lysine and arginine.

Specify the intensity cutoff to remove low intensity signal. For the example dataset (i.e., analyzed with Spectronaut 19 and acquired using an Orbitrap Fusion Lumos platform) we recommed to use a cutoff of 8. This filtering can be performed in the quantification step in Spectronaut (from 19.3 onwards). Note, removing these extremely low values leads to a dramatic improvement of H/L ratio quantification in early time points of a pSILAC experiment.

The `requant` and `inpute.method` parameters are only relevant when working with OpenSwath results in long format.

See `?generatepSILACObject` for a full documentation.

3.2 Examples

3.2.1 Spectronaut 19 example without replicate design

- Use example data provided in the KdeggeR package.

```
input_data <- KdeggeR::example_spectronaut
input_design <- KdeggeR::example_spectronaut_design

pSILAC_object <- KdeggeR::generatepSILACObject(dataset = input_data,
                                              design = input_design,
                                              inputDataType = "spectronaut",
                                              aggregate.replicates = NA, # replicates not aggregated
                                              filterPeptides = T,
                                              ncores = NULL,
                                              noiseCutoff = 8)

pSILAC_object$info
pSILAC_object$design
```

3.2.2 Spectronaut 19 example with replicate aggregation

- Use example data provided in the KdeggeR package.
- The replicates will be averaged during the pSILAC object generation.
- This design is advisable when processing experiments with, e.g., replicate injections or dish replicates.

```
input_data <- KdeggeR::example_spectronaut
input_design <- KdeggeR::example_spectronaut_design_replicates

pSILAC_object <- KdeggeR::generatepSILACObject(dataset = input_data,
                                              design = input_design,
                                              inputDataType = "spectronaut",
                                              aggregate.replicates = "mean", # or "median"
                                              filterPeptides = T,
                                              ncores = NULL,
                                              noiseCutoff = 8)

pSILAC_object$info
pSILAC_object$design
```

3.2.3 Spectronaut 19 example, use filepath

- The files are provided in the inst/data_tsv directory via the GitHub repository.
- The function will load the files using the provided paths and process the data.

```
# design without replicates
input_data <- paste(.libPaths()[1],
                    "KdeggeR/data_tsv/example_data_pSILAC_SN19.tsv",
                    sep = "/")
input_design <- paste(.libPaths()[1],
                      "KdeggeR/data_tsv/example_design_table_pSILAC_SN19_no_replicates.txt",
                      sep = "/")

pSILAC_object <- KdeggeR::generatepSILACObject(dataset = input_data,
                                   design = input_design,
                                   inputDataType = "spectronaut", # needs to be specified
                                   aggregate.replicates = NA, # replicates not aggregated
                                   filterPeptides = T,
                                   ncores = NULL,
                                   noiseCutoff = 8)

# design with replicates
input_design <- paste(.libPaths()[1],
                      "KdeggeR/data_tsv/example_design_table_pSILAC_SN19_replicates.txt",
                      sep = "/")

pSILAC_object <- KdeggeR::generatepSILACObject(dataset = input_data,
                                   design = input_design,
                                   inputDataType = "spectronaut", # needs to be specified
                                   aggregate.replicates = "mean", # or "median"
                                   filterPeptides = T,
                                   ncores = NULL,
                                   noiseCutoff = 8)
```

3.2.4 DIA-NN 1.9 example

- Use example data provided in the KdeggeR package.

```
input_data <- KdeggeR::example_diann
input_design <- KdeggeR::example_diann_design

pSILAC_object <- KdeggeR::generatepSILACObject(dataset = input_data,
                                   design = input_design,
                                   inputDataType = "diann",
                                   aggregate.replicates = NA,
                                   filterPeptides = T,
                                   ncores = NULL,
                                   noiseCutoff = 8)

pSILAC_object$info
pSILAC_object$design
```

3.2.5 DIA-NN 1.9, use filepath

- The files are provided in the inst/data_tsv directory via the GitHub repository.
- The function will load the files using the provided paths and process the data.

```
# design without replicates
input_data <- paste(.libPaths()[1],
                    "KdeggeR/data_tsv/example_data_pSILAC_diann_1,9.tsv",
                    sep = "/")
input_design <- paste(.libPaths()[1],
                      "KdeggeR/data_tsv/example_design_table_pSILAC_diann_1,9.tsv",
                      sep = "/")

pSILAC_object <- KdeggeR::generatepSILACObject(dataset = input_data,
                                   design = input_design,
                                   inputDataType = "diann",
                                   aggregate.replicates = NA,
                                   filterPeptides = T,
                                   ncores = NULL,
                                   noiseCutoff = 8)
```

4 Filter data

We recommend to apply several data filtering steps before model fitting. These filtering steps are optional for the downstream analysis, but improve the results of the modeling using both non-linear least squares (NLS) fitting and linear modeling (lm) of the precursor and protein turnover rates. In principle, the data do not need any filtering based on valid values, since the model can still estimate rates using a single data point. However, in a dataset with a low level of missing values, these lower potentially quality fits can be removed.

We first filter data based on valid values and then based on the correct trend, following the assumption that the H/L ratios should be increasing over the time points in a steady-state pSILAC experiment.

These can be wither performed for the whole time-series or skipping the first time point if very short and expected to be noisy. For example, in a 4-5 time point experiment (such as 1, 4, 8, 12, and 24 hours), we would request at least two valid values in the later time points skipping the first, noisy data point, filter the remaining precursors based on correct trend, and then focus on keeping only high quality data in the first, 1 hour time point.

To do so, we replace by NA all values in time point 1, which do not follow the expected trend in this time point, and which were detected as significant outlier from a fit predicted by linear regression of the ln(H/L + 1) data.

4.1 Filter based on valid values

- Filter based on valid values, skip the first time point (preferable for the example dataset).

```
pSILAC_object <- KdeggeR::filterValidValues(pSILAC_object,
                                            values_cutoff = 2,
                                            skip_time_point = 1)
```

- Filter based on valid values, at least 3 values requested in a time-series.

```
# example code, does not modify the pSILAC object
KdeggeR::filterValidValues(pSILAC_object,
                           values_cutoff = 3,
                           skip_time_point = 0)
```

4.2 Filter based on monotone trend

- Removes precursors which do not follow the expected trend, skipping time point 1 (preferable for the example dataset).

- Replaces values in time point 1, which do not follow the expected trend considering the next time point available.

```
pSILAC_object <- KdeggeR::filterMonotone(pSILAC_object,
                                         skip_time_point = 1)

pSILAC_object <- KdeggeR::filterMonotoneTimePoint1(pSILAC_object)
```

- Removes precursors which do not follow the expected trend, including time point 1.

```
# example code, does not modify the pSILAC object
KdeggeR::filterMonotone(pSILAC_object,
                        skip_time_point = 0)
```

4.3 Filter based on linear regression

- Applies linear regression to the ln(H/L + 1) data and Grubbs' test to detect outliers in the first time point.
- Time point 1 values with a Grubbs' test P value lower than a selected threshold (`p_cutoff`) and corresponding to curves with an R2 lower than selected threshold (`R2_cutoff`) will be replaced by NA in both ln(H/L + 1) data and RIA data.

```
pSILAC_object <- KdeggeR::filterLinearRegression(pSILAC_object,
                                                 skip_time_point = 1,
                                                 R2_cutoff = 0.9,
                                                 p_cutoff = 0.05)
```

5 Fit models to estimate k_loss

5.1 The RIA method

At each time point, the amount of heavy (H) and light (L) precursor was extracted and used to calculate the relative isotopic abundance RIAt.

- RIA_t=L/(L+H)

The value of RIAt changes over time as unlabeled proteins are gradually replaced by heavy-labeled proteins throughout the experiment. This occurs because of cell division, which dilutes the unlabeled proteins, and the natural turnover of intracellular proteins, where the loss rate can be described by an exponential decay process.

- RIA_t=RIA_0 .e ^((-k_loss . t))

Where RIA0 denotes the initial isotopic ratio and kloss the rate of loss of unlabeled protein. We assumed RIA0 = 1, as no heavy isotope was present at t = 0, thus the value of RIAt will decay exponentially from 1 to 0 after infinite time and used nonlinear least-squares estimation to perform the fit. As discussed before, these assumptions may reduce measurement error, especially at the beginning of the experiment, where isotopic ratios are less accurate.

```
pSILAC_object <- KdeggeR::calcRIAkloss(pSILAC_object,
                                       ncores = NULL)
```

5.2 The NLI method

A simpler approach to determine de facto protein degradation rates is to directly calculate the rate of loss from the light peptide intensities. The light peptide intensities need to be normalized using median channel sums to calculate the normalized intensity values (NLI), which is done during the pSILAC object generation step.

Then, the light precursor rate of loss can be modeled using the same model and assumptions as in the case of the RIA-based modeling. As we reported previously, the NLI and RIA method results are strongly correlated, however, the NLI method tends to have higher variability. However, since the low-abundant heavy signals are not required, this method might provide more precursor-level k_loss and if desired, the results can be combined with or complement the RIA-based k_loss values during the protein level aggregation.

If the data are already filtered, the `startIntensity` of `max` can be used for the modeling, as the intensity in the first measured time point is the maximum one. Other options are available and might be used for unfiltered data, such as `median` or `model`, see the function documentation.

```
pSILAC_object <- KdeggeR::calcNLIkloss(pSILAC_object,
                                       startIntensity = "max",
                                       ncores = NULL)
```

5.3 The H/L method

The heavy proteins are synthesized over time, leading to an increasing H/L ratio. This process is exponential because the heavy proteins are gradually replacing the unlabeled (light). The H/L ratios are linearized by log-transformation and the rate of incorporation of the heavy label is then estimated from a linear model.

- ln(H/L+1)= k_syn .t

In the steady-state condition, the rates of protein synthesis and degradation reach equilibrium. This means that the rate at which new heavy-labeled proteins are synthesized must be balanced by the rate at which proteins are degraded or turned over (kloss).

```
pSILAC_object <- KdeggeR::calcHoLkloss(pSILAC_object,
                                       tryRobust = FALSE,
                                       ncores = NULL)
```

6 Calculate protein k_loss

Protein-level k_loss values can be calculated by different options including performing a weighted average of the selected fit (e.g., RIA only) or their combination/complement (e.g., RIA and NLI). The number of data points used to estimate precursor-level k_loss, the variance of the fit, or both can be used as weights.

The `method` can be set either to `RIA`, `hol`, or `NLI`, which uses only the selected precursor-level k_loss to calculate protein level k_loss. Alternatively, the `combined` and `complement` methods use both RIA-based and NLI-based k_loss. The `combined` method selects the most stable estimate between `RIA` and `NLI`, while the `complement` method primarily uses the RIA-based estimates and complements them with the NLI-based estimates, when the RIA-based are missing. A `source` column in the output protein k_loss table indicates whether RIA or NLI-based calculation was used for a specific protein.

The `ag.metric` can be set as either `mean` or `median`. If `mean`, the protein-level k_loss values are calculated as a weighted average. The weights can be specified using the `ag.weights` parameter as `nbpoints`, `variance`, or `both`.

The `returnKlossTableOnly` parameter controls whether the protein k_loss values are saved in the pSILAC object or exported as a data.frame in the R enviroment. Optionally, the standard deviations can be reported by setting the `returnSD` parameter to TRUE.

See the function doumentation `?calcProteinsKloss` for more details.

6.1 Perform weighted average

```
pSILAC_object <- KdeggeR::calcProteinsKloss(pSILAC_object, method = "complement",
                        ag.metric = "mean",
                        ag.weights = "both",
                        ncores = NULL,
                        returnKlossTableOnly = F,
                        returnSD = F )
```

6.2 Direct export of protein-level k__loss

The `returnKlossTableOnly = T` will directly return the protein k_loss data.frame rather than updating the pSILAC object.

```
prot_kloss <- KdeggeR::calcProteinsKloss(pSILAC_object, method = "RIA",
                        ag.metric = "mean",
                        ag.weights = "both",
                        ncores = 1,
                        returnKlossTableOnly = T, # export protein k_loss df
                        returnSD = T )
```

7 Calculate all rates

The `calcAllRates()` function is a wrapper function that calc the individual functions above to perform precursor k_loss calculation using the RIA, ln(H/L +1), and NLI data.

Then it aggregates the precursor-level estimates into protein k_loss estimates using the selected method. It accepts the same parameters as the functions above.

```
pSILAC_object <- KdeggeR::calcAllRates(pSILAC_object,
                                       method = "complement",
                                       ag.metric = "mean",
                                       ag.weights = "both")
```

8 Calculate k__deg and t__1/2

Protein degradation rates are estimated by subtracting the cell division rates (kcd) to correct for the protein pool dilution caused by the exponential cell division.

- k_deg=k_loss-k_cd

However, practically, the cell division rates tend to be very variable between different experiments and thus the precision and accuracy tend to be low. Therefore, we enabled the option to use a k_cd derived from the distribution of the k_loss values by assuming that most k_deg values should be positive after the correction. We suggest a value (k_perc) by subtraction of which only 1% of k_deg values would be negative.

- k_deg=k_loss-k_perc

Optionally, protein half-lives from the degradation rate constant using the following formula. - t_(1/2) = (ln(2))/k_deg

8.1 Import experimentally-determined kcd values

```r
input_kcd <- KdeggeR::example_kcd
```

8.2 Calculate kdeg using kcd

To use the experimentally-derived `kcd` value set the `type` parameter to `kcd` and provide a data.frame with the k_cd values using the `rate_df` parameter.

```r
pSILAC_object <- KdeggeR::calcKdeg(pSILAC_object,
                                   rate_df = input_kcd,
                                   type = "kcd")
```

8.3 Calculate kdeg using kperc

To use the theoretical `kperc` value set the `type` parameter to `kperc`. The percentage of negative k_deg values after the k_perc subtraction can be set using the `perc_neg` parameter. Based on our experience, setting this paramater to a value between 0.01 and 0.05 performs the best, and we also observed such as percentage of negative k_deg values when we subtracted the experimentally-derived k_cd values.

```r
pSILAC_object <- KdeggeR::calcKdeg(pSILAC_object,
                                   rate_df = NULL,
                                   type = "kperc",
                                   perc_neg = 0.01)
```

8.4 Calculate t(1/2)

The halflives are calculated using the formula described above.

```r
pSILAC_object <- KdeggeR::calcHalflife(pSILAC_object)
```

9 Visulize Results

9.1 Precursor/peptide

Plot the peptide-level fit of a selected peptide. The colors can be specified in the design table and customized by modifing the design data.frame of the pSILAC object.

9.1.1 Plot precursor RIA model

```
pSILAC_object$peptides %>%
  dplyr::slice_head(n = 10) %>%
  dplyr::glimpse()

# Spectronaut example
KdeggeR::plotPeptideRIA(pSILAC_object,
                        peptide = "_MLIPYIEHWPR_.3")
```

9.1.2 Plot precursor ln(H/L +1) model

```
KdeggeR::plotPeptideHoL(pSILAC_object,
                        peptide = "_MLIPYIEHWPR_.3")
```

9.2 Protein

Plot protein-level k_loss fit and all precursor-level datapoints.

9.2.1 Plot protein RIA

```
KdeggeR::plotProteinRIA(pSILAC_object,
                        protein = "O75208")
```

9.2.2 Plot protein HoL

```
KdeggeR::plotProteinHol(pSILAC_object,
                        protein = "O75208")
```

9.2.3 Plot protein summary

Plots both RIA-based and ln(H/L +1)-based protein k_loss fit and peptide-level k_loss distribution as boxplots.

```
KdeggeR::plotProtein(pSILAC_object,
                     protein = "O75208")
```

10 Statistical analysis

Statistical analysis of the example dataset can be performed using the `limma` package after log2 transformation and valid values filtering of the k_deg values.

In the example dataset, a cisplatin-resistant cell line, A2780Cis (“Cis”), is compared to a cisplatin-sensitive, parental cell line, A2780 (“Nor”).

10.1 Transform and filter k_deg data

```r
# select how to filter data based on valid values
n_replicates <- 2 # at least 2 replicates

prot_kdeg <- pSILAC_object$protein.kdeg %>%
  # log2 transform
  dplyr::mutate(across(.cols = everything(),
                       ~suppressWarnings(log2(.)))) %>%
  # calculate valid values per condition
  tibble::rownames_to_column("id") %>%
  dplyr::rowwise() %>%
  dplyr::mutate(
    valid_Cis = sum(!is.na(dplyr::c_across(contains("Cis"))) &
                    !is.infinite(dplyr::c_across(contains("Cis")))),
    valid_Nor = sum(!is.na(dplyr::c_across(contains("Nor"))) &
                    !is.infinite(dplyr::c_across(contains("Nor"))))
  ) %>%
  dplyr::ungroup() %>%
  dplyr::glimpse() %>%
  # filter based on valid values
  dplyr::filter(valid_Cis >= n_replicates &
                  valid_Nor >= n_replicates) %>%
  dplyr::select(-starts_with("valid_")) %>%
  as.data.frame() %>%
  tibble::column_to_rownames("id") %>%
  dplyr::glimpse()
```

10.2 Statistical analysis - moderated t-test

The statistical analysis in this examples is performed using the `limma` package.

10.2.1 Load the package

```r
library(limma)
```

10.2.2 Generate condition matrix

```r
condition_matrix <- data.frame(sample_name = colnames(prot_kdeg)) %>%
  dplyr::mutate(condition = gsub("_\\d{1}$", "", sample_name))
```

10.2.3 Generate design matrix

```r
design_matrix <- model.matrix(~0 + condition,
                              data = condition_matrix)

colSums(design_matrix)
```

10.2.4 Generate contrast matrix

```
contrast_matrix <- makeContrasts(Cis_Nor = conditionCis - conditionNor,
                    levels = design_matrix)
contrast_matrix
```

10.2.5 Fit linear model into the data

```
# Fit linear model for each protein
fit.linear <- limma::lmFit(prot_kdeg,
                           design_matrix)

# Given a linear model fit to the data,
# compute estimated coefficients and standard errors
# for a given set of contrasts.
fit.test <- limma::contrasts.fit(fit.linear,
                                 contrasts = contrast_matrix)
```

10.2.6 Compute moderated statistics using eBayes

```
# Fit linear model for each protein
fit.linear <- limma::lmFit(prot_kdeg,
                           design_matrix)

# Given a linear model fit to the data,
# compute estimated coefficients and
# standard errors for a given set of contrasts
fit.test <- limma::contrasts.fit(fit.linear,
                                 contrasts = contrast_matrix)

# Given a linear model fit from lmFit, compute moderated t-statistics,
# moderated F-statistic, and log-odds of differential expression by
# empirical Bayes moderation of the standard errors towards a global value.
fit.test <- limma::eBayes(fit.test)

# results table containing all proteins
results <- limma::topTable(fit.test,
                                adjust.method = "none",
                                p.value = 1,
                                lfc = log2(1),
                                number = nrow(prot_kdeg))

results <- results %>%
  tibble::rownames_to_column("id")
```

10.2.7 Example volcano plot

```
results %>%
  dplyr::mutate(
    significant = ifelse(
      adj.P.Val < 0.05 & logFC > log2(1.5), "upregulated",
      ifelse(adj.P.Val < 0.05 & logFC < -log2(1.5), "downregulated", "nonregulated")
    )
  ) %>%
  ggplot(aes(x = logFC, y = -log10(adj.P.Val),
             color = significant,
             size = significant,
             alpha = significant)) +
  geom_point(shape = 4) +
  theme_light() +
  labs(title = "A2780Cis x A2780") +
  scale_color_manual(values = c("steelblue3", "grey80", "gold1")) +
  scale_alpha_manual(values = c(0.8, 0.5, 0.8)) +
  scale_size_manual(values = c(1, 0.5, 1)) +
  geom_hline(yintercept = -log10(0.05), color = "darkred",
             linetype = "dashed") +
  geom_vline(xintercept = c(log2(1 / 1.5), log2(1.5)),
             color = "darkred", linetype = "dashed") +
  theme(plot.title = element_text(hjust = 0.5))
```