Package 'proximetricsR'

June 30, 2026

Type Package

Title Spectral Preprocessing and Chemometric Calibration of NIR Sensors

Version 0.6.4

Date 2026-06-22

Maintainer Leonardo Ramirez-Lopez <ramirez-lopez.l@buchi.com>

BugReports https://github.com/l-ramirez-lopez/proximetricsr/issues

Description Provides tools to build quantitative chemometric models and applications for near-infrared (NIR) sensors. Chemometric regression models are based on partial least squares regression as described by Wold (1975) <doi:10.1016/B978-0-12-103950-9.50017-4> and modified partial least squares regression as described by Shenk and Westerhaus (1991) <doi:10.2135/cropsci1991.0011183X003100020049x>, with further discussion by Westerhaus (2014) <doi:10.1255/nirn.1492>.

License MIT + file LICENSE

URL https://github.com/l-ramirez-lopez/proximetricsr

VignetteBuilder quarto

Depends R (>= 4.2.0)

Imports callr, digest (>= 0.6), foreach, mathjaxr (>= 1.0), plotly (>= 4.0), prospectr (>= 0.2.10), quarto, uuid, withr, zip, readxl, jsonlite, Rcpp

Suggests knitr, testthat, covr, doParallel, parallel, devtools

LinkingTo Rcpp, RcppArmadillo

RdMacros mathjaxr

NeedsCompilation yes

LazyData true

LazyDataCompression xz

Encoding UTF-8

Config/testthat/edition 3

Config/VersionName Saentis

Config/roxygen2/version 8.0.0

Config/roxygen2/markdown TRUE

Author Leonardo Ramirez-Lopez [aut, cre] (ORCID:
<https://orcid.org/0000-0002-5369-5120>),
Claudio Orellano [aut] (ORCID: <https://orcid.org/0009-0005-7523-4236>),
Nicolae Cudlenco [aut] (ORCID: <https://orcid.org/0000-0001-6547-3659>),
Mai Said [aut] (ORCID: <https://orcid.org/0000-0001-6979-8725>),
Mohamed Abushosha [aut],
Marcal Plans [aut] (ORCID: <https://orcid.org/0000-0001-9894-2626>)

Repository CRAN

Date/Publication 2026-06-30 20:30:07 UTC

Contents

`proximetricsR-package` *Overview of the proximetricsR package*

Description

NIR calibration and application tools for BUCHI ProxiMate and ProxiScout devices.

Details

This is package version 0.6.4 (Saentis).

This package provides `R` functions for spectral pre-processing, NIR model calibration, and reading/writing files for BUCHI ProxiMate and ProxiScout devices. The calibration algorithms (`fit_plsr`, `fit_xlsr`) and the pre-treatment constructors (`prep_smooth`, `prep_snv`, `prep_resample`, `prep_derivative`) reproduce the corresponding algorithms in BUCHI NIRWise PLUS (version 1.1.3000.0), guaranteeing numerical compatibility between models built with this package and those built in NIRWise PLUS.

The ProxiScout functions for preprocessing are also numerically equivalent to the ones of the "BUCHI Modeller" software. The regression method in te Modeller is teh classical PLS regression, however, the other PLS algorithms implemented in proximetricsR (modified PLS, standard PLS, and XLS) can also be used to generate models for ProxiScout devices.

The functions available for ProxiMate spectral data are:

- `proximate_read_data`
- `proximate_data`
- `proximate_merge`

The functions available for reading generic spectral data files are:

- `read_spc`

The functions available for spectral pre-processing are:

- `prep_resample`
- `prep_smooth`
- `prep_snv`
- `prep_derivative`
- `prep_detrend`
- `prep_transform`

- prep_wav_trim
- preprocess_recipe
- process

The functions available for calibrating NIR regression models are:

- calibrate
- calibrate_models
- calibration_control
- fit_plsr
- fit_xlsr
- add_model_metadata
- validate_prediction

The functions available for writing ProxiMate files are:

- proximate_write_data
- proximate_write_model
- add_application_metadata
- proximate_write_nax

The functions available for reading and editing ProxiMate application files are:

- proximate_read_cal
- proximate_read_nax
- proximate_recalibrate_nax
- proximate_add2nax

The functions available for ProxiScout devices are:

- proxiscout_read_data
- proxiscout_write_data
- proxiscout_write_model
- get_proxiscout_wavenumbers
- proxiscout_repetition_pattern

The functions available for creating plots are:

- plot.spectral_model

Other functions:

- extract_property_names

A typical example dataset for a ProxiMate device can be found in:

- NIRcannabis

Author(s)

Leonardo Ramirez-Lopez, Claudio Orellano, Nicolae Cudlenco, Mai Said, Mohamed Abushosha, Marcal Plans

See Also

Useful links:

- https://github.com/l-ramirez-lopez/proximetricsr
- Report bugs at https://github.com/l-ramirez-lopez/proximetricsr/issues

---

`add_application_metadata` *A function for adding application metadata to a list of* `spectral_model` *objects*

Description

This function has two use cases:

i. If `object` (a list of `spectral_model` objects) is passed to the function, it returns the same object with the specified application metadata added to it.

ii. Otherwise, the function can be used to create a list of application metadata that can be used as input for the argument `metadata` of the `proximate_write_nax` function.

Usage

```
add_application_metadata(
  object, key = UUIDgenerate(),
  name = c(name = "Untitled", alias = NULL),
  view = c("Up", "Down"), measurement_mode = c("DrIwr", "TrIwr"),
  measurement_time = 15,
  absorbmask_low = c(min = 0, max = 0),
  absorbmask_high = c(min = 0, max = 0),
  rotate_sample = TRUE,
  selectable = TRUE, created, changed,
  composition = NULL,
  description = "created with proximetricsR",
  sop = "",
  presentation_id = "Default"
)
```

Arguments

object	an optional object, consisting of a list of objects of class spectral_model. See details.
key	a string for the key of the application. Defaults to a newly generated key using UUIDgenerate.
name	a vector length at most 2, consisting of characters for the name and alias of the application. Defaults to "Untitled".
view	a string for the type of view in the application. Has to be either "Up" (default) or "Down".
measurement_mode	a string, indicating how the samples were measured. Has to be either Diffuse Reflection ("DrIwr", default) or Transflection ("TrIwr").
measurement_time	a numeric for the time each sample in the application should be measured, in seconds. Defaults to 15 seconds.
absorbmask_low	a vector of numerics of length 2 for the minimum and maximum of the lower absorbance mask. Defaults to a vector of zeros.
absorbmask_high	a vector of numerics of length 2 for the minimum and maximum of the higher absorbance mask. Defaults to a vector of zeros.
rotate_sample	a logical. Should the sample be rotated? Defaults to TRUE.
selectable	a logical, whether the application should be selectable. Defaults to TRUE.
created	a string of date and time of the creation of the application. Default is the current date and time of the system. See details for the format in which it has to be provided.
changed	a string of date and time when the application was changed. Defaults to the current date and time of the system. See details for the format in which it has to be provided.
composition	an optional string for the composition of the application. Defaults to NULL.
description	an optional string for the description of the application. Defaults to "created with proximetricsR".
sop	a string for the standard operating procedure (sop) for this particular application. Defaults to an empty character.
presentation_id	a string for the sample presentation ID of the application. Default is "Default".

Details

This function has two functionalities:

- If object (a list of spectral_model objects) is passed to the function, it returns the same object with the specified application metadata added to it.
- Otherwise, the function can be used to create a list of application metadata that can be used as input for the argument metadata of the proximate_write_nax function.

The application metadata is required for the import of an application into a ProxiMate device.

The two-fold functionality of this function allows to add application metadata during the construction of the models, or after the model-building processes have been finished. In the former case, a list of models of class `spectral_model` must be passed in `object`. Then, the returned object of this function contains the same list of models, including the specified metadata. Models can also be added or removed from that list, without changing the application metadata. In the latter case, the returned value of this function may be passed to the parameter `metadata` of function `proximate_write_nax`.

A lot of the parameters can be left unchanged and may be adjusted at a later stage of the application development (e.g. in a ProxiMate device). However, several parameters are of great importance for a successful migration of the application:

The parameter `view` describes if the spectrum is measured by either up-view `"Up"` or down-view `"Down"`.

The `measurement_mode` describes how the samples are measured, with the following possibilities: Diffuse Reflection `"DrIwr"` or Transflection `"TrIwr"`.

The parameters `created` and `changed` must contain the date (`YYYY-MM-DD`) and time (`HH:MM:SS`), seperated by a single `"T"` (without any spaces). For example, the following code returns the correct format (both `created` and `changed` default to this value):

```
gsub(" ", "T", format(Sys.time()))
```

Value

Either the list of `spectral_model` objects with the added application metadata (if `object` is provided), or the application metadata as a named list.

Author(s)

Claudio Orellano, Leonardo Ramirez-Lopez

See Also

`calibrate`, `proximate_write_nax`

Examples

```
data(NIRcannabis)

# Downview Absorbance of CBDA in percentage
downview_metadata <- add_application_metadata(
  name = "CBDA Downview",
  view = "Down",
  measurement_mode = "DrIwr"
)

# Create a simple model with default model metadata
simple_model <- calibrate(CBDA ~ spc,
  data = NIRcannabis, preprocess = preprocess_recipe(),
  method = fit_plsr(5), control = calibration_control(),
  metadata = add_model_metadata(), verbose = FALSE
```

```
)

# Two ways to add application metadata to a list of spectral_model objects:
model_list <- list(simple_model)

# Using the add_application_metadata 'object' argument
model_list <- add_application_metadata(
  object = model_list,
  name = "CBDA Downview",
  view = "Down",
  measurement_mode = "TrIwr"
)

# Adding it manually
model_list$metadata <- downview_metadata

# Alternatively, if you are creating an application, you can also pass
# application metadata to 'proximate_write_nax':
proximate_write_nax(
  object = model_list,
  path = tempdir(),
  metadata = downview_metadata,
  tsv_name = "some_tsv",
  empty_tsv_name = "another_tsv",
  report = TRUE,
  verbose = FALSE
)
```

`add_model_metadata` *A function for adding model metadata to a* `spectral_model` *object*

Description

This function has two use cases:

i. If `object` (being a `spectral_model` object) is passed to the function, it returns the same object with the specified model metadata added to it.

ii. Otherwise, the function creates a a list of model metadata that can be used as input for the argument `metadata` of the `calibrate` function.

Usage

```
add_model_metadata(
  object, key = UUIDgenerate(), created, changed,
  name = c("", NULL), sort_order = 1, tol_min = NULL,
  tol_max = NULL, decimal_places = 2, unit = "",
  mahal_limit = 5, corrections = c(bias = 0, slope = 1),
  limit_min = NULL, limit_max = NULL, target = NULL,
  wavelength_range = c("Nir", "Vis", "Nir+Vis"),
```

```
  predict_type = "Calibration", arguments = rep("", 4)
)
```

Arguments

object	an optional object of class spectral_model. See details.
key	a string for the key of the model. Defaults to a newly generated key using UUIDgenerate.
created	a string for date and time of the addition of the model to the application. Default is the current date and time of the system. See details for the format in which it has to be provided.
changed	a string for date and time when the model has been changed. Default is the current date and time of the system. See details for the format in which it has to be provided.
name	a vector of character strings of length 2 for the name and alias of the property. If object is given or an object returned by this function is passed to calibrate, defaults to the name of the property (but not the alias). Otherwise, defaults to an empty character.
sort_order	a numeric, indicating the order in which the properties are shown on a ProxiMate device. Defaults to 1.
tol_min	an optional numeric for the minimum error tolerance. Defaults to NULL.
tol_max	an optional numeric for the maximal error tolerance. Defaults to NULL.
decimal_places	a numeric for the decimal precision of the measurements of the property. Defaults to 2.
unit	a string for the units in which the reference values of the property are measured. Defaults to an empty character.
mahal_limit	a numeric for the maximum Mahalanobis distance allowed. Defaults to 5.
corrections	a vector of numerics of length 2 for bias and slope corrections. Defaults to no corrections, i.e. c(0, 1).
limit_min	an optional numeric for the lower limit of the reference values. Defaults to NULL.
limit_max	an optional numeric for the upper limit of the reference values. Defaults to NULL.
target	an optional numeric for the desired predicted reference values. Defaults to NULL.
wavelength_range	a string for the considered wavelength range of the spectrum. Must be one of "Nir" (default), "Vis" or "Nir+Vis".
predict_type	a string for the prediction type of the model. Defaults to "Calibration".
arguments	a vector of maximal length 4. Contains additional arguments to be saved into the metadata. Defaults to a vector of empty characters of length 4.

Details

This function has two functionalities:

- If object (being a spectral_model object) is passed to the function, it returns the same object with the specified property metadata added to it.

- Otherwise, the function creates a a list of property metadata that can be used as the argument `metadata` of the `calibrate` function.

The two-fold functionality of this function allows to add metadata during the construction of the model, or after the model-building has been finished. For the former, the model has to be passed in `object`, and the returned value of this function contains the model including the chosen metadata. In the latter case, the returned value of this function may be passed to the parameter `metadata` of function `calibrate`.

A lot of the parameters can be left unchanged and may be adjusted at a later stage of the application development (e.g. in a ProxiMate device).

The parameters `created` and `changed` must contain the date (`YYYY-MM-DD`) and time (`HH:MM:SS`), seperated by a single `"T"` (without any spaces). For example, the following code returns the correct format (also, both `created` and `changed` default to this value):

```
gsub(" ", "T", format(Sys.time()))
```

Value

Either the `spectral_model` object with the added property metadata (if `object` is provided), or the property metadata, which is a named list.

Author(s)

Claudio Orellano, Leonardo Ramirez-Lopez

See Also

`calibrate`, `proximate_write_nax`

Examples

```
data(NIRcannabis)

# Downview Absorbance of CBDA in percentage
downview_metadata <- add_model_metadata(
  name = "CBDA",
  unit = "%",
  arguments = "Example metadata"
)

# Three ways to add metadata to spectral_model object:
# As a direct argument
simple_model <- calibrate(CBDA ~ spc,
  data = NIRcannabis, preprocess = preprocess_recipe(),
  method = fit_plsr(5), control = calibration_control(),
  metadata = downview_metadata
)

# Passing the model to add_model_metadata
simple_model <- add_model_metadata(
  object = simple_model,
```

```
  name = "CBDA",
  unit = "%",
  arguments = "Example metadata"
)

# Adding it directly (not recommended)
simple_model$metadata <- downview_metadata
```

`calibrate` *Calibrate a spectral model*

Description

Produce calibrations for predictive partial least squares (pls) or extended partial least squares (xls) models using cross-validation and outlier detection. Reproduces the modeling methods in NIRWise PLUS calibration software.

Usage

```
## S3 method for class 'formula'
calibrate(formula, data, group = NULL,
          preprocess = preprocess_recipe(prep_snv()),
          method,
          metadata = NULL,
          return_inputs = TRUE,
          ...,
          na_action = na.pass)

## Default S3 method:
calibrate(X, Y, data = NULL, group = NULL,
          preprocess = preprocess_recipe(prep_snv()),
          method = fit_plsr(ncomp = min(15, dim(X))),
          control = calibration_control(),
          metadata = NULL,
          skip_indices = NULL,
          return_inputs = TRUE,
          verbose = TRUE,
          ...)

## S3 method for class 'spectral_model'
predict(object, newdata, ncomp = object$final_ncomp,
          verbose = TRUE, ...)
```

Arguments

`...`	not currently used.
`formula`	an object of class `formula` which represents the basic model to be calibrated.

data	a data.frame containing the data of the variables in the model. Must be provided if using S3 method for class formula. Otherwise, optional; however, if using proximate_write_nax for the returned object, this parameter will be required.
X	a numeric matrix of spectral data. The names of the columns must be equivalent to wavelengths, such that they can be coerced to class numeric.
Y	a matrix of one column with the response variable. The column must be named.
group	an optional factor (or character vector that can be coerced to factor by as.factor) that assigns a group/class label to each observation in X (e.g. groups can be given by spectra collected from the same batch of measurements, from the same observation, from observations with very similar origin, etc). This is taken into account for cross-validation for pls tuning (factor optimization) to avoid pseudo-replication. When one observation is selected for cross-validation, all observations of the same group are removed and assigned to validation. The length of the vector must be equal to the number of observations in X.
preprocess	a preprocess_recipe object as returned by the preprocess_recipe function, indicating the pretreatments to be applied on the spectra before the regression steps.
method	an object of class fit_constructor, as returned by fit_plsr or fit_xlsr, indicating what type of regression method to use along with its parameters.
control	a calibration_control object as returned by the calibration_control function, indicating how some aspects of the calibration process must be conducted (e.g. cross-validation and outlier detection).
metadata	either NULL or an object as returned by method add_model_metadata. Contains the specifications for the metadata of the model. Defaults to NULL.
skip_indices	a vector of integers for the indices in the input data to be skipped for the regression. Defaults to NULL
return_inputs	a logical. For calibrate methods, indicates if the input data should be attached to the returned object. Note that this data is crucial for creating an application file.
verbose	a logical indicating whether or not to print a progress bar for the iterations of the validation along with messages of the execution of the cross-validation. For the predict method, messages about the progress are printed. Default is TRUE. Note: In case parallel processing is used, these progress bars are not printed.
object	an object of class spectral_model.
newdata	a data.frame containing the new spectral data of the variables in the model, of similar form as data. Alternatively, can also be a matrix of spectra.
ncomp	a vector for the number of components to be used in the prediction. Default is object$final_ncomp i.e. the optimized number of components found in the object passed to predict.
na_action	a function to specify the action to be taken if NAs are found in the object passed in data. Default is na.pass.

Details

The resulting object of the `calibrate` functions provides a complete list of calibration results.

By using the `group` argument one can specify groups of observations that have something in common (e.g. observations with very similar origin). The purpose of `group` is to avoid biased cross-validation results due to pseudo-replication. This argument allows to select calibration points that are independent from the validation ones. In this regard, the `p` argument used in object passed to `control` (and created with the `calibration_control` function), refers to the percentage of groups of observations (rather than single observations) to be retained in each sampling iteration.

The regression algorithms implemented here correspond to the partial least squares ("pls") and extended partial least squares ("xls") methods in NIRWise PLUS calibration software. Note that in these particular regression algorithms, the Y-loading of each component is constantly equal to 1, and therefore not considered.

The `calibration_statistics` matrix retrieved in the `final_model` and also in the `initial_fit` outputs includes a column named `Q_value`. This value can be used to asses model overfitting. For each observation, q_i is computed as follows:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(y_j - \hat{y}_j)^2}{n-1}}$$

$$q_i = \frac{|2y_i - \hat{y}_i - \ddot{y}_i|}{s}$$

where for ith observation, y is the observed value, $\hat{y}$ is the fitted value (using a model with all the observations) and $\ddot{y}$ is the predicted value during cross-validation.

Value

For `calibrate()`, an object of class `spectral_model` which is a list with the following elements:

- `formula`: The formula used (only output if the S3 method for class `'formula'` was used).
- `dataclasses`: The data classes in the model (only output if the S3 method for class `'formula'` was used).
- `target_variable`: A character for the name of the target/response variable for which the predictive model was built.
- `predictor_variables`: A character vector for names of the predictor variables (wavelengths) used to build the model.
- `final_model`: A list with:
  - `model_cv`: A list of cross-validation results.
  - `ncomp`: The number of components used for the model. If cross-validation is used, this is the optimal number of components for the chosen tuning parameter and learning rates (see `calibration_control`).
  - `model`: An object of class `spectral_fit`. See `spectral_fit` for the full structure.
  - `calibration_statistics`: A matrix showing the prediction statistics for each calibration sample for the optimal number of components used in the model (if cross-validation is used, see `calibration_control`). It contains the following columns:

    * Sample_index: The indices of the samples.
    * Target: The target/response variable of the samples.
    * fitted_y: The fitted values of the model of each sample. This row is equivalent to the row of the optimal component of fitted_y inside the fitted model in model.
    * residual: The residuals of the fitted values of each sample. Note that the residuals are obtained as the difference of targets and fitted values.
    * predicted_y_in_cv: The predicted values as computed in the cross-validation. Only available for k-fold and leave-one-out cross-validation.
    * cv_residual: The residuals of the predicted values of the cross-validation. Only available for k-fold and leave-one-out cross-validation.
    * Mahalanobis: The squared Mahalanobis distance of each sample in the score space to the origin.
    * Q_value: The Q-value of each sample. See details
  - calibration_statistics_all: A list of matrices with the same information as in calibration_statistics, but for all components.
  - detected_outliers_all: A list of lists, each containing the same information as in the detected_outliers$model_* mentioned below, but for all components in the fitted model.
- detected_outliers: A named list, containing the following entries:
  - model_*: A named list, containing all detected outliers of the particular model, identified based on the calibration residual limit ("calibration"), the Mahalanobis distance limit ("Mahalanobis"), and the validation residual limit ("validation"). The number of such model_* entries depends on the number selected in remove_outliers of the control argument; if it is selected to be 0, then only one model is fitted, so only model_1 exists; for higher choices of remove_outliers, the number of models of this list is at most remove_outliers + 1: for every time a model is fitted, a new entry in the detected_outliers is generated.
  - all: A named list, containing all detected outliers of all models produced, similarly to model_*. In particular, this entry is the combination of all detected outliers in the model_* entries of the list, where the specific type of outlier is retained.
  - removed: A single vector, containing all removed outliers of the final model. This vector is empty whenever the remove_outliers of the control argument is set to 0 or if no outlier has been found. Otherwise, this vector is a combination of all different outliers that were removed whenever a new model has been fitted, while ignoring the specific type of the outlier. In particular, in case the last model still contains at least one outlier, this vector is a combination of all but the last entry of the model_* lists. If the last fitted model does not contain any outlier, this vector is a combination of all model_* lists, and hence the vectorized form of the all entry of the list.

  See calibration_control for more information on the limits and the outlier removal procedure.
- initial_fit: A list similar to final_model, but before any outliers were removed. Only stored if outlier removal is requested (i.e. remove_outliers in the control argument is larger than 0). In that case, the model here contains only the very first model that was fitted without any detected outliers removed.
- final_ncomp: An integer, indicating the final/optimal number of components to be used.

- preprocess: A preprocess_recipe object mirroring the input of the preprocess argument.
- processed_wavs: A processed_wavs object providing the spectral variables that existed in the data right before each preprocessing step.
- method: A fit_constructor object mirroring the input of the method argument.
- control: A calibration_control object mirroring the input of the control argument.
- preprocessed_X: The preprocessed spectral data for the observations of the final model. Spectra with missing values, skipped indices and removed outliers are discarded from the matrix.
- skipped_indices: A list with two objects:
  - missing_response: A vector of indices of observations with missing response values.
  - manually_skipped: A vector of indices mirroring the input of the skip_indices argument.
- input_data: A list, which is only returned if return_inputs is set to TRUE. Mirrors the input of the data argument.

For predict(), the output is an object of class spectral_prediction, which is a list with the following elements:

- predictions: A matrix with the predictions of the response variable using the new spectral data (newdata), based on the provided model (object). Contains only the predictions of the requested number of components (ncomp).
- scores: A matrix with the projected new data onto the score space of the provided model. Contains the scores of all possible number of components.
- model_information: A list, containing information on the model input of object:
  - target_var: A character, indicating the name of the target variable.
  - preprocess_recipe: A character, indicating the spectral preprocessing recipe and its order.
  - model_grid: A matrix, containing the grid of the model object, such as the coefficient of determination and the RMSE of the validation for the requested number of components.
  - unit: A character, indicating the units of the model.
  - opt_comp: An integer, signifying the optimal number of components as computed by the validation process of the model.

Parallel cross-validation

The cross-validation loop is implemented with foreach, so it can be parallelised transparently by registering a parallel backend before calling calibrate. Set allow_parallel = TRUE in calibration_control (the default) and register a backend, for example:

```
cl <- parallel::makeCluster(parallel::detectCores() - 1L)
doParallel::registerDoParallel(cl)

model <- calibrate(...)

parallel::stopCluster(cl)
```

When no parallel backend is registered, `foreach` falls back silently to sequential execution regardless of the `allow_parallel` setting. Note that progress bars are suppressed during parallel execution.

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

`preprocess_recipe`,

`fit_plsr`,

`fit_xlsr`,

`calibration_control`,

`calibrate_models`

Examples

```
data("NIRcannabis")
simple_model <- calibrate(CBDA ~ spc,
  data = NIRcannabis, preprocess = preprocess_recipe(prep_snv()),
  method = fit_xlsr(5), control = calibration_control("kfold"),
  verbose = FALSE
)

method <- fit_plsr(15)
control <- calibration_control(validation_type = "kfold", number = 3, folds = "sequential")
pretreats <- preprocess_recipe(
  prep_resample(grid = c(1001, 1700, 5)),
  prep_derivative(m = 2, w = 9, p = 5, algorithm = "nwp"),
  prep_snv(),
  prep_smooth(w = 5, algorithm = "moving-average"),
  device = "proximate"
)
skip_indices <- c(5, 13, 21, 73)
# With formula
complex_model_formula <- calibrate(
  CBDA ~ spc,
  data = NIRcannabis, preprocess = pretreats, method = method,
  control = control, skip_indices = skip_indices, verbose = FALSE
)
# Default, need care with Y
Y <- matrix(NIRcannabis$CBDA)
colnames(Y) <- "CBDA"
complex_model_default <- calibrate(
  X = NIRcannabis$spc, Y = Y, data = NIRcannabis, preprocess = pretreats,
  method = method, control = control, skip_indices = skip_indices, verbose = FALSE
)

# Predict the skipped indices
predict(complex_model_formula,
```

```
  newdata = NIRcannabis[skip_indices, ],
  ncomp = complex_model_formula$final_ncomp,
  verbose = FALSE
)
```

`calibrate_models` *Calibrate models for multiple response variables*

Description

Calibrate independent models (iteratively) for multiple properties with optimization of both the preprocessing recipe (based on a list of different recipes) and the regression method. This function uses `calibrate` to construct such list of models.

Usage

```
calibrate_models(
  formulas,
  data, group = NULL,
  preprocess_recipes,
  methods,
  control = calibration_control(seed = 1),
  metadata_list = NULL,
  skip_indices_list = NULL,
  return_inputs = TRUE,
  ...,
  na_action = na.pass,
  verbose = TRUE,
  save_all = FALSE
)

## S3 method for class 'spectral_multimodel'
predict(object, newdata, verbose = TRUE, ...)
```

Arguments

`formulas`	a list containing one or more objects of class `formula` where each of them represents the model to be calibrated.
`data`	a data.frame containing the data of the variables in the model (as in the `calibrate` function).
`group`	an optional factor (or character vector that can be coerced to `factor` by `as.factor`) that assigns a group/class label to each observation in `X` (e.g. groups can be given by spectra collected from the same batch of measurements, from the same observation, from observations with very similar origin, etc). This is taken into account for cross-validation for pls tuning (factor optimization) to avoid pseudo-

replication. When one observation is selected for cross-validation, all observations of the same group are removed and assigned to validation. The length of the vector must be equal to the number of observations in X.

preprocess_recipes
: a list with one or more objects of class preprocess_recipe that are to be tested for finding the optimal one for each model in the list passed to formulas.

methods
: a list containing one or more objects of class fit_constructor as returned by fit_plsr or fit_xlsr, indicating what type of regression method to use along with its parameters.

control
: a calibration_control object as returned by the calibration_control function, indicating how some aspects of the calibration process must be conducted (e.g. cross-validation and outlier detection). Default is calibration_control(seed = 1). See details.

metadata_list
: a list containing the specifications for the metadata of each model in formulas given in the same order. Each element in the list should be defined as in the metadata argument of calibrate using the add_model_metadata function. Defaults to NULL.

skip_indices_list
: a list of vectors of integers for the indices in the input data to be skipped for the computation of each of the models in formulas. The vectors in this list must be provided in the same order as their corresponding counterparts in formulas. Defaults to NULL. In case a list is passed, the list components must be filled with numeric() for those formulas where there is no indices to be skipped.

return_inputs
: a logical. For calibrate methods, indicates if the input data should be attached to the returned object. Note that this data is crucial for creating an application file.

...
: arguments to be passed to the calibrate method. Not currently used for the predict.spectral_multimodel method.

na_action
: a function to specify the action to be taken if NAs are found in the object passed in data. Default is na.pass.

verbose
: a logical indicating whether or not to print a progress bar for the iterations of the validation along with messages of the execution of the cross-validation. For the predict method, messages about the progress are printed. Default is TRUE. Note: In case parallel processing is used, these progress bars are not printed.

save_all
: a logical indicating if all the models tested (with the different pre-processing recipes) are to be saved. Default is FALSE.

object
: an object of class spectral_multimodel.

newdata
: a data.frame containing the new spectral data of the variables in the model, of similar form as data. Alternatively, can also be a matrix of spectra.

Details

The object passed to the control argument should indicate a seed for the random number generator (RNG). This allows the function to use the same cross-validation validation groups (for leave group-out cross-validation, see calibration_control) across the same formula with different recipes. This enables proper model comparisons.

Value

A list of class "spectral_multimodel" containing the following objects:

- results_grid: a data.frame with the validation results of the best models found for each preprocessing recipe with the best regression method applied on the spectral data of the model built for each formula.
- all_models: if save_all, a list with the spectral_model objects corresponding to all the models tested.
- final_models: a list containing only the spectral_model objects corresponding to the best models found for each formula. This list can be used/passed later to the proximate_write_nax function to produce an application file (in that case it might be convenient to add some metadata to the resulting models in the list using the add_model_metadata function).

For predict(), a list with the following elements:

- predictions: A matrix with the predictions of the response variable using the new spectral data (newdata), based on the provided models (object). Contains only the predictions of the optimal number of components (ncomp).
- model_information: A list, containing information on the models inputs in object. Each component in the list contains the following information:
  - target_var: A character, indicating the name of the target variable.
  - preprocess_recipe: A character, indicating the spectral preprocessing recipe and its order.
  - model_grid: A matrix, containing the grid of the model object, such as the coefficient of determination and the RMSE of the validation for the requested number of components.
  - unit: A character, indicating the units of the model.
  - opt_comp: An integer, signifying the optimal number of components as computed by the validation process of the model.

Parallel cross-validation

The cross-validation loop inside each call to calibrate is implemented with foreach, so it can be parallelised transparently by registering a parallel backend before calling calibrate_models. Set allow_parallel = TRUE in calibration_control (the default) and register a backend, for example:

```
cl <- parallel::makeCluster(parallel::detectCores() - 1L)
doParallel::registerDoParallel(cl)

result <- calibrate_models(...)

parallel::stopCluster(cl)
```

When no parallel backend is registered, foreach falls back silently to sequential execution regardless of the allow_parallel setting. Note that progress bars are suppressed during parallel execution.

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

calibrate,

preprocess_recipe,

fit_plsr,

fit_xlsr,

calibration_control

Examples

```
data("NIRcannabis")
# the list of formulas for the models to be built
app_formulas <- list(THC ~ spc, THCA ~ spc, CBD ~ spc, CBDA ~ spc)

# the list of pre-processing recipes to be tested
precipes <- list(
  recipe_1 = preprocess_recipe(
    prep_resample(grid = c(1001, 1700, 2)),
    prep_snv(),
    prep_derivative(m = 1, w = 9, p = 7, algorithm = "nwp"),
    device = "proximate"
  ),
  recipe_2 = preprocess_recipe(
    prep_resample(grid = c(1001, 1700, 2)),
    prep_snv(),
    prep_derivative(m = 2, w = 11, p = 9, algorithm = "nwp"),
    device = "proximate"
  )
)

optimized_app <- calibrate_models(
  formulas = app_formulas,
  data = NIRcannabis,
  preprocess_recipes = precipes,
  methods = list(fit_plsr(15, type = "nwp")),
  return_inputs = TRUE,
  save_all = FALSE
)

optimized_app
```

`calibration_control` *A function that controls the calibration of models*

Description

This function is used to further control some aspects of the calibration of models (with the `calibrate` function) such as cross-validation and outlier detection.

Usage

```
calibration_control(validation_type = c("lgo", "loo", "kfold", "none"),
                    number = ifelse(validation_type == "lgo", 100, 10),
                    p = 0.75,
                    folds = c("random", "sequential"),
                    tuning_parameter = c("rmse", "rsq", "none"),
                    learning_rates = c(maximum = 1.1, sequential = 1.05),
                    remove_outliers = 0,
                    cal_residual_limit = 2.5,
                    mahalanobis_limit = 5,
                    val_residual_limit = 3.5,
                    allow_parallel = TRUE,
                    fix_pls_factors = TRUE,
                    fixed_components = 0,
                    replacements = TRUE,
                    seed = NULL)
```

Arguments

`validation_type`
: a character string indicating the type of cross-validation (cv) to be conducted. Options are: `"lgo"` for leave-group-out cv (default), `"loo"` for leave-one-out cv and `"kfold"` for k-fold cv. See details.

`number`
: an integer indicating the number of sampling iterations or sub-sample groups for the selected `validation_type` argument. Default is `100` for leave-group out cv and `10` for k-fold cross-validation. This parameter is ignored for leave-one-out cv.

`p`
: a numeric value indicating the percentage of calibration observations to be retained at each sampling iteration at each local segment when `"lgo"` is selected in the `validation_type` argument. Default is 0.75 (i.e. 75 percent of the observations).

`folds`
: a character string indicating the way folds are created (valid only when `validation_type = "kfold"`). Options are: `"random"` (default) or `"sequential"`.

`tuning_parameter`
: a character string indicating which cross-validation statistic to use for the optimization of the included number of components. Options are: `"rmse"` (default, minimization of the root mean squared error), `"rsq"` (maximization of

the coefficient of determination) or "none" (no tuning). Does not apply when validation_type = "none".

learning_rates a vector of length 2 for additional control over the selection of the optimal number of components. See details for its use. Defaults to c(1.1, 1.05).

remove_outliers
an integer indicating the number of times the model should automatically detect and remove outliers. Each time, a new model is fitted with the outliers removed, until either no more outliers are found or the remove_outliers has been reached. Outliers found and removed in each step, as well as the first and last computed models are recorded. Outliers are detected based on the limits set in the arguments cal_residual_limit, mahalanobis_limit and val_residual_limit. Setting remove_outliers to 0 (default) disables automatic outlier removal, whereas selecting it as Inf removes outliers until no more are found.

cal_residual_limit
a numeric value which indicates the upper limit of the standardized residuals for the fitted response variable. Observations with absolute residuals above this limit are labeled as "calibration outliers". The standardized calibration residuals are calculated as the absolute differences between the reference values and their corresponding fitted values divided by the standard deviation of these absolute differences. Default is 2.5 (as in NIRWise PLUS calibration software).

mahalanobis_limit
a numeric value which indicates the upper limit of the squared Mahalanobis distances of each sample in the score space to zero. Observations with squared Mahalanobis distance above this limit are labeled as "Mahalanobis outliers". The squared Mahalanobis distances are calculated as the squared Euclidean distance of the standardized scores to the origin. Default is 5 (as in NIRWise PLUS calibration software).

val_residual_limit
a numeric value which indicates the upper limit of the standardized residuals for cross-validation predictions of the response variable. This applies only to "kfold" or "loo" cross-validation. Observations with absolute residuals above this limit are labeled as "validation outliers". The standardized validation residuals are calculated as the absolute differences between the reference values and their corresponding cross-validated predictions divided by the standard deviation of these absolute differences. Default is 3.5 (as in NIRWise PLUS calibration software).

allow_parallel a logical indicating if parallel execution is allowed. If TRUE, parallelization is applied to the cross-validation procedure. The parallelization of this for loop is implemented using the foreach function of the foreach package. Default is TRUE.

fix_pls_factors
a logical. This parameter only has an influence on the produced application files, where it indicates whether the final number of factors of the model should be fixed. Note that this has no influence on the model in R itself, as the optimal number of components inside the model remains the same (but it does influence the exported files). Default is TRUE.

`fixed_components`
: a numerical value indicating a fixed number of components to be used in the model (i.e. no optimization of the components). The default value is `0`, which indicates that the number of components is not fixed and it uses the one selected by the function.

`replacements`
: a logical. Only used in case `validation_type` is selected as `"lgo"`. Specifies if the sampling for the calibration sets must be done with replacements. See details for a more thorough explanation. Defaults to `TRUE`.

`seed`
: an integer that can be used in any of the validation methods to obtain reproducible results, using the `set.seed` function. In case it is selected as `NULL`, no seed will be set. Note that the seed will not be reset, and future random computations can be affected. Furthermore, this parameter is meant as a way to provide reproducibility, and should not be used to simply select the seed with the best results. Default is `NULL`.

Details

This package extends the cross-validation methods implemented in the NIRWise PLUS software, which is based only on k-fold cross validation.

The validation methods available for assessing the predictive performance of the models are:

- **Leave-group-out cross-validation (`"lgo"`):** The data is partitioned into different subsets of similar size. Each partition is based on a stratified random sampling using the distribution of the response variable. When `p` $>= 0.5$ (i.e. the number of calibration observations to retain is larger than 50% of the total samples), the sampling is conducted for selecting the validation samples, and when `p` < 0.5 the sampling is conducted for selecting the calibration samples (samples used for model training). The model fitted with the selected calibration samples is used to predict the target response variable values of the validation samples. The accuracy and precision, indicated by the root mean square error (RMSE) and the coefficient of determination (R^2) respectively, are computed. This process is repeated m times (where m is controlled by the `number` argument), and the final RMSE and R^2 are computed as the average over all respective results of the m iterations. In case the parameter `replacements` is set to `TRUE`, the selection of the calibration sets is done by using sampling with replacement.
- **Leave-one-out cross-validation (`"loo"`):** The number of iterations is equal to the number of observations in the calibration set. In each iteration, one single observation is held out, while the remaining samples are used to fit a model, which is used to predict the response variable of the held out observation. The predictions are then compared to the reference ones and both the RMSE and the (R^2) are computed.
- **k-fold cross-validation (`"kfold"`):** The data is split (either randomly or sequentially) into k disjoint blocks of similar size, where k is controlled by `number`. In the sequential splits, every block B_i is selected as follows:

$$B_i = \{i + k(j-1) | j \in N,\ i + k(j-1) \leq n\}$$

where n is the total number of observations. In other words, the observations are put sequentially into the blocks until all observations have a block assigned.
A total of k iterations is conducted. In each iteration, one block is considered as the validation set, while the remaining samples are used to fit a model, which is then used to predict the

response variable of the held-out block.
The number observations in each block is given by the total number of observations divided by the number of blocks. Note that the maximum number of folds is limited to half of the number of observations. Note also that this implementation of k-fold cross-validation is an improved version of the one in the NIRWise PLUS software, where only the sequential sample selection is supported.

- **No validation (`"none"`):** No validation is carried out.

For each validation type (except `"none"`), the optimal number of factors is not necessarily chosen to be the minimum of RMSE or the maximum of R^2 (depending on the `tuning_parameter`). Instead, since both are often monotonically decreasing respectively monotonically increasing as the number of components increases, an additional parameter `learning_rates` γ for fine-tuning of the determination of the number of factors is included:

For RMSE, consider the index where the minimum of all computed RMSE is attained:

$$n_{min} = arg\min_n \ RMSE_n$$

,

Then, among all $1 < n < n_{min}$ fulfilling

$$RMSE_n < RMSE_{n_{min}} \cdot \gamma_{max}$$

$$RMSE_n < RMSE_{n+1} \cdot \gamma_{seq}$$

we take the smallest n as the optimal number of components.
For R^2, a similar approach is taken, but with maxima instead of minima: $n_{max} = arg\max_n RMSE_n$
Then, take the smallest $1 < n < n_{max}$ still satisfying

$$R^2_n > R^2_{n_{max}} \cdot \gamma^{-1}_{max}$$

$$R^2_n > R^2_{n+1} \cdot \gamma^{-1}_{seq}$$

Note that in this case, we take the inverse of the learning rates. Furthermore, setting `learning_rates = c(1, 1)` retains the global minimum for RMSE, respectively maximum for R^2.

Value

a list of class `calibration_control` mirroring the specified parameters

Author(s)

Leonardo Ramirez-Lopez

See Also

`calibrate`, `calibrate_models`

Examples

```
# 5-fold cross-validation with sequential sampling
calibration_control(
  validation_type = "kfold",
  number = 5,
  folds = "sequential"
)

# leave-one-out cross_validation
calibration_control(validation_type = "loo")

# 100 leave-group-out validations with 60% samples retained, with replacements
calibration_control(
  validation_type = "lgo",
  number = 100,
  p = 0.6,
  replacements = TRUE
)

# 2-fold leave-group-out cross-validation with 75% samples retained, no replacements
calibration_control(
  validation_type = "lgo",
  number = 2,
  p = 0.75,
  replacements = FALSE
)

# Same as before, but removing any outlier that is found
calibration_control(
  validation_type = "lgo",
  number = 2,
  p = 0.75,
  replacements = FALSE,
  remove_outliers = Inf
)

# no validation, gives warning
calibration_control(validation_type = "none")
```

`extract_property_names` *Extract the property names from a given* `data.frame`

Description

This function aims to extract the column names of properties from `x`. A property in this context is a response vector of numerical values that then later can be calibrated for predictions (such as with `calibrate`).

Usage

```
extract_property_names(x)
```

Arguments

x	a data.frame, as normally obtained by proximate_read_data, read_spc, proxiscout_read_data, or some other data parsing function.

Details

Depending on the class of x, the names of the properties are identified differently. For all cases, only columns which contain numerical values (including NA) are considered as potential properties.

If x is of class proximate_data, the property names are identified as follows:

- Located between columns "Reference" and "Begin".
- Not named according to any of the following names: "ROW", "Check", "Date", "SNR", "SRN", "ID", "Barcode", "Note", "Result", "Reference", "Begin", "End", "Recipe", "Composition", "Images", "spc".
- Contain only numerical values (including NA).

If x is of class proxiscout_data, property names are identified as columns that contain only numerical values (including NA) and are not matched by any of the following, case-insensitive regex (each wrapped by ^ and $):

- id
- sample[_.]?name
- captured[_.]?at
- device[_.]?id
- created[_.]?(by|at)
- on[_.]?behalf[_.]?of
- lot[_.]?name
- scanner([_.]?id)?
- original[_.]?value
- display[_.]?value
- note
- location
- supplier
- device
- spc
- predictions

If x is of neither class, all columns with numerical values are considered to be properties

Value

A character vector, containing only the names of numerical properties. If no property names were identified, return a character vector of length 0.

`fit_constructors` *Fitting method constructors*

Description

These functions create configuration objects that specify the regression method to be used within `calibrate`.

Usage

```
fit_plsr(ncomp, type = c("nwp", "standard", "modified"))

fit_xlsr(ncomp, type = c("nwp", "standard", "modified"), min_w = 3, max_w = 15)
```

Arguments

`ncomp`	a positive integer indicating the maximum number of PLS components to use.
`type`	a character string indicating the algorithm variant. One of `"nwp"` (default), `"standard"`, or `"modified"`. • `"nwp"`: replicates the NIRWise PLUS method, which uses correlation-based weights with an additional slope correction applied to the weights and scores. • `"standard"`: standard PLS using standardised covariances between spectra and reference values as weights. • `"modified"`: modified PLS using correlations between spectra and reference values as weights.
`min_w`	a positive integer indicating the minimum window size for the XLS algorithm. Default is `3`.
`max_w`	a positive integer indicating the maximum window size for the XLS algorithm. Must be greater than `min_w`. Default is `15`.

Details

There are two regression methods available:

Partial least squares (`fit_plsr`): Uses PLS regression. The only parameter optimised is the number of components (`ncomp`). Three algorithm variants are available via `type`: `"nwp"`, `"standard"`, and `"modified"`.

Extended partial least squares (`fit_xlsr`): Uses the XLS algorithm. In addition to `ncomp` and `type`, the window range (`min_w`, `max_w`) controls the local smoothing applied within the algorithm.

Value

An object of class `c("fit_plsr", "fit_constructor")` or `c("fit_xlsr", "fit_constructor")` containing the specified parameters, to be passed to `calibrate`.

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

calibrate, calibrate_models

Examples

```
# PLS as in NIRWise PLUS
fit_plsr(ncomp = 15)

# Standard PLS with 15 components
fit_plsr(ncomp = 15, type = "standard")

# Modified PLS with 15 components
fit_plsr(ncomp = 15, type = "modified")

# XLS as in NIRWise PLUS
fit_xlsr(ncomp = 10)

# Standard XLS with custom window range
fit_xlsr(ncomp = 10, type = "standard", min_w = 5, max_w = 20)
```

---

get_proxiscout_wavenumbers

ProxiScout standard wavenumbers

Description

Returns the standard wavenumbers used by ProxiScout NIR scanners.

Usage

```
get_proxiscout_wavenumbers()
```

Details

The standard wavenumbers of ProxiScout (see https://www.si-ware.com/) NIR scanners range from approximately 3921.569 cm^{-1} to 7407.407 cm^{-1} in steps (resolution) of around 13.61655 cm^{-1}. This is equivalent to a spectral range of 1350 to 2550 nm, with a varying resolution that starts from 2.486189 nm at 1350 nm and ends with a resolution of 8.823525 nm at 2550 nm.

Value

A numeric vector containing the standard wavenumbers of ProxiScout NIR scanners.

Examples

```
# Get the complete set of ProxiScout wavenumbers
wavs <- get_proxiscout_wavenumbers()

# Get the corresponding wavelengths (nm)
wavelengths_nm <- 10000000 / wavs

# Display the range of wavenumbers
range(wavs)
```

NIRcannabis *NIRcannabis*

Description

Selected samples of cannabis NIR measurements for demo purposes. The dataset contains absorbance spectra of 80 cannabis samples measured between 1001 nm and 1700 nm at a 3 nm interval. A total number of four reference vectors is included: `"CBDA"` (Cannabidiolic acid), `"THCA"` (Tetrahydrocannabinolic acid), `"CBD"` (Cannabidiol) and `"THC"` (Tetrahydrocannabinol).

Usage

```
data("NIRcannabis")
```

Format

A `data.frame` containing 80 observations of four response variables, with their corresponding spectral data.

Details

This dataset is an example for a typical data file for ProxiMate applications, with a total of 80 cannabis samples, selected as a subset of a larger database. It contains the following rows for each observation:

- `ROW`**:** Integers for the associated numbers inside the database.
- `Check`**:** Characters, indicating whether the particular observation should be included in the construction of the model inside a ProxiMate.
- `Date`**:** Characters for the date and time when the measurement was taken.
- `SNR`**:** Characters of the serial number of the involved ProxiMate device.
- `ID:` Characters for the ID's.
- `Barcode`**:** Characters for the barcodes.
- `Notes`**:** Characters for the notes.
- `Result`**:** Characters for the results.
- `Reference`**:** Characters containing all reference values, concatented into one character with semicolon separation.

- `CBDA`: Numerics for the reference values of Cannabidiolic acid.
- `THCA`: Numerics for the references values of Tetrahydrocannabinolic acid.
- `CBD`: Numerics for the reference values of Cannabidiol.
- `THC`: Numerics for the reference values of Tetrahydrocannabinol.
- `Begin`: Characters, indicating when the measuring was initiated.
- `End`: Characters, indicating when the measurement was completed.
- `Recipe`: Characters for the recipe.
- `Composition`: Characters for the composition of the sample.
- `Images`: Characters for the image of the samples.
- `spc`: A numerical matrix of the absorbance spectra, corresponding to each individual observation.

Source

BUCHI Labortechnik AG.

`plot.spectral_model` *Plot results of a given model*

Description

Create a html file for a number of useful analytical plots using the R Quarto file "model_plot_template.qmd" for the given model `x` of class `spectral_model`.

Usage

```
## S3 method for class 'spectral_model'
plot(
  x, validations = NULL, output_file = x$target_variable,
  output_dir = NULL,
  spectral = c("weights", "coefficients", "scores", "mahalanobis"),
  cv = c("error", "response", "residuals", "qq", "distributions"),
  regression = NULL,
  validation = if (!is.null(validations)) "all" else NULL,
  verbose = TRUE, open_file = TRUE, ...
)
```

Arguments

`x`	an object of class `"spectral_model"`. This model should be generated using the `calibrate` function.
`validations`	an optional object of class `"spectral_validation"`. This object, if provided, should be generated using `validate_prediction`. Default is `NULL`.

output_file	a character string for the name of the generated file. Default is the target name saved in model x.
output_dir	a string for the directory in which the file is generated. Default is NULL, which writes the file to tempdir().
spectral	a character vector of spectral plots to include, "all" to include every spectral plot, or NULL to skip the section entirely. Available names: "raw", "preprocessed", "weights", "loadings", "coefficients", "scores", "scores_3d", "scaled_scores", "mahalanobis".
cv	a character vector of cross-validation plots to include, "all" to include every CV plot, or NULL to skip the section entirely. Available names: "error", "response", "response_overview", "residuals", "qq", "q_values", "distributions".
regression	a character vector of regression analysis plots to include, "all" to include every regression plot, or NULL (default) to skip the section entirely. Available names: "response", "response_overview", "residuals", "qq", "residuals_vs_fitted", "scale_location", "leverage".
validation	a character vector of validation plots to include, "all" to include every validation plot, or NULL to skip. Available names: "predicted_vs_reference". Defaults to "all" when validations is supplied, NULL otherwise.
verbose	a logical. When TRUE (default), prints the path of the generated file. Pandoc output is always suppressed.
open_file	a logical, indicating whether the file should automatically be opened in a browser after compilation. Defaults to TRUE.
...	additional graphical parameters. See details.

Details

This function creates a html file from rendering the R Markdown file 'model_plot_template.qmd' using quarto::quarto_render(). This will generate an .html file with the given output_file as its name in the directory specified by output_dir. Note that any existing file in the given directory of similar name will be overwritten.

The file opens automatically in the default browser of the system if open_file is set to TRUE.

Depending on the size of the provided dataset, the produced file might take a long time to process, and the files can quickly get quite large. The four section arguments (spectral, cv, regression, validation) control which plots are included. Each accepts a character vector of plot names, "all" to include the entire section, or NULL to skip it. For example, to render every available plot:

```
plot(x, spectral = 'all', cv = 'all', regression = 'all',
     validation = 'all')
```

The available plots per section are as follows (defaults marked with *):

spectral

- **Raw Spectra**: A line plot of all raw spectra. Only available if input data is saved inside the model x, i.e. if the method calibrate was called with return_inputs is set to TRUE. Note that the depicted spectrum always has a resolution of 10.

- **Preprocessed spectra**: A line plot of all preprocessed spectra. Note that the depicted spectrum always has a resolution of 10.
- **Weights***: A line plot of all weights.
- **Loadings**: A line plot of all loadings.
- **Coefficients***: A line plot of all regression coefficients.
- **Scores***: A points plot of scores for each component.
- **3D Scores**: A three dimensional points plot of scores for each component. The component for the x-axis can be selected with a slider. The corresponding y- and z-axis are the previous and next component, respectively.
- **Scaled Scores**: A points plot of the scaled scores for each component.
- **Mahalanobis Distance***: A points plot of the Mahalanobis distance of the scaled scores of each component.

cv

Only available if the calibration used cross-validation. For leave-group-out cross-validation, only `"error"` is available.

- **Error measures***: A plot of error and precision measures. In particular, this plot depicts the largest residual, the RMSE and the R- squared measures for the cross-validation for all components. The optimal component is highlighted.
- **CV Response Plot***: A points plot of the reference values versus the cross-validation predictions made by the model for each component. Additionally, the identity line is added, plus a regression line fitted with the use of the a linear regression model.
- **CV Response Plot Overview**: An overview of all CV Response Plot in a single plot.
- **CV Residuals***: A points plot of the residuals of the cross-validated predictions, for every component.
- **Q-Q Plot of CV Residuals***: A Q-Q plot of the sample quantiles of the standardized cross-validated residuals against the theoretical quantiles of a normal distribution for each component. A line with intercept zero and slope one is depicted.
- **CV Q-Values**: A points plot of the Q-values of the cross- validation in the model for each component. See details of calibrate for an explanation of the Q-values.
- **Distributions***: A line plot of the densities of the reference values and the cross-validated predictions for each component.

regression

These plots do not necessarily indicate model performance - more components generally improve fit but may overfit. Useful for identifying outliers. Similar to `plot.lm`.

- **Response Plot**: A points plot of the reference values versus the fitted values for each component. Additionally, the identity line is added and a regression line is fitted using a linear regression model.
- **Response Plots Overview**: An overview of all Response plots in a single plot.
- **Residuals**: A points plot of the residuals of the fitted values for each component.

- **Q-Q Plot of Residuals**: A Q-Q plot of the sample quantiles of the standardized residuals against the theoretical quantiles of a normal distribution for each component. A line with intercept zero and slope one is depicted.
- **Residuals vs Fitted**: A points plot of the fitted values against their residuals for each component. Additionally, a line for the LOESS smoother is depicted.
- **Scale Location Plot**: A points plot of the fitted values against the square roots of the absolute values of the standardized residuals for each component. Additionally, a line for the LOESS smother is depicted.
- **Leverage vs Residuals**: A points plot of the leverages of the fitted values against the standardized residuals for each component. Additionally, a line for the LOESS smother is depicted.

validation

Only available when `validations` is supplied (an object of class `spectral_validation` from `validate_prediction`).

- **Predicted vs. Reference***: Shows the predictions of the new data obtained from the model versus the actual reference values, with an identity line, plus a regression line fitted with the use of a linear regression model. Additionally, the R^2 and `RMSE` of both the validated predictions and model is depicted. See `validate_prediction` and `predict` for more details on the prediction and validation process.

Most of above plots contain a slider, which may be used to adjust the considered component. The sliders start at the optimal components (if any calibration control was applied) or at the maximum number of components (otherwise).

The plots are constructed with the help of the plotly package. As such, the possibilities to manipulate the plots are as in that package. The arrangement of the plots is controlled by the quarto package.

Additional graphical parameters may be supplied to this function by using the ellipsis argument `...`. These arguments will be passed to some of the scatter and layout functions of plotly. More precisely, the arguments are passed to possible attributes of `add_trace`, and `layout` function of plotly. However, the following arguments will always be ignored:
`c("p", "sliders", "x", "x0", "dx", "y", "y0", "dy", "visible", "type", "name","hovertext", "text", "mode")`,
as well as arguments passable to both `add_trace`, and `layout`. The `"line"` attribute is ignored when plotting markers and vice-versa. Some plots ignore the ellipsis argument altogether.

Possible attributes of these functions may be found by using the function `schema` of plotly.

Value

NULL. The desired plots are opened in a browser window.

Author(s)

Claudio Orellano, Leonardo Ramirez-Lopez

Examples

```
data("NIRcannabis")
control <- calibration_control(validation_type = "kfold", number = 3, folds = "sequential")
prepro_recipe <- preprocess_recipe(
  prep_resample(grid = c(1001, 1700, 2)),
  prep_snv(),
  prep_derivative(m = 1, w = 9, p = 7, algorithm = "nwp"),
  device = "proximate"
)
skips <- c(5, 13, 21, 73)
my_model <- calibrate(CBDA ~ spc,
  data = NIRcannabis, preprocess = prepro_recipe,
  method = fit_plsr(15), control = control, skip = skips, verbose = FALSE
)

plot(my_model, output_dir = tempdir())
# Include every available plot in every section
plot(my_model, output_dir = tempdir(),
  spectral = "all", cv = "all", regression = "all", validation = "all"
)
# Custom section selection
plot(
  my_model,
  output_file = "example_plot",
  output_dir = tempdir(),
  spectral = c("weights", "scores"),
  cv = "all",
  regression = NULL
)
# Make predictions and validate
preds <- predict(my_model, NIRcannabis[skips, ])
validations <- validate_prediction(preds, NIRcannabis$CBDA[skips])
# Plot validation section only
plot(
  my_model,
  output_dir = tempdir(),
  output_file = "example_plot",
  validations = validations,
  spectral = NULL,
  cv = NULL,
  regression = NULL
)
```

---

preprocess_recipe *Build and execute spectral preprocessing recipes*

Description

The preprocess_recipe function assembles an ordered sequence of preprocessing steps into a recipe, while process executes the recipe on a spectral data matrix.

Usage

```
preprocess_recipe(..., device)

process(X, recipe, device)
```

Arguments

`...`	one or more objects of class `preprocessing` as returned by any of the following constructor functions: • `prep_resample` • `prep_smooth` • `prep_snv` • `prep_derivative` • `prep_detrend` • `prep_transform` • `prep_wav_trim` The order in which the objects are provided defines the order of execution. If no arguments are provided, an empty recipe is returned and `process` will return the input data unchanged.
`device`	a character string specifying the target device: `"unspecified"` (no validation), `"proximate"`, or `"proxiscout"`. When `"proximate"` or `"proxiscout"` is specified, `preprocess_recipe` validates that all steps are compatible with that device and raises an informative error if not. Pass `"unspecified"` to skip validation explicitly. `device` is required whenever the recipe contains any preprocessing step, with one exception: a recipe containing only a single `prep_snv` step does not require `device`, because SNV is device-agnostic (identical behaviour for both `"proximate"` and `"proxiscout"`). In that case `device` defaults to `"unspecified"`.
`X`	a numeric matrix of spectral data to be preprocessed (samples in rows, wavelengths in columns).
`recipe`	an object of class `preprocess_recipe` as returned by `preprocess_recipe`. A single object of class `preprocessing` is also accepted and treated as a one-step recipe.

Value

For `preprocess_recipe`, an object of class `preprocess_recipe` with three components: `steps` (the ordered list of preprocessing step objects), `device` (the target device string), and `preprocessing_order` (a simplified string summarising the sequence of applied transformations).

For `process`, a numeric matrix of preprocessed spectral data. The applied recipe is stored as the attribute `"preprocess_recipe"` on the returned matrix and can be retrieved with `attr(result, "preprocess_recipe")`.

Author(s)

Leonardo Ramirez-Lopez

See Also

prep_smooth, prep_snv, prep_derivative, prep_resample, prep_detrend, prep_transform, prep_wav_trim

Examples

```
data("NIRcannabis")
X <- NIRcannabis$spc

# SNV alone — no device needed (SNV is device-agnostic)
recipe_snv <- preprocess_recipe(prep_snv())
X_snv <- process(X, recipe_snv)

# Any other combination requires device
recipe <- preprocess_recipe(
  prep_smooth(w = 7, p = 1, algorithm = "savitzky-golay"),
  prep_snv(),
  prep_derivative(m = 1, w = 5, p = 2, algorithm = "savitzky-golay"),
  device = "proxiscout"
)

X_proc <- process(X, recipe)
attr(X_proc, "preprocess_recipe")
```

---

prep_derivative *Derivative constructor for spectral preprocessing*

Description

Creates a preprocessing constructor for computing first or second order derivatives of spectral data. The constructor is intended to be passed to preprocess_recipe and executed via process.

Three algorithms are supported: Savitzky-Golay ("savitzky-golay"), Norris-Gap/Gap-Segment ("gap-segment"), and the derivative pre-treatment from BUCHI NIRWise PLUS software ("nwp").

Usage

```
prep_derivative(m, w, p, algorithm = c("savitzky-golay", "gap-segment", "nwp"))
```

Arguments

m	An integer indicating the derivative order. Must be 1 (first derivative) or 2 (second derivative).
w	A positive odd integer indicating the filter window size. For "gap-segment", w indicates the gap size (spacing between points over which the derivative is computed).
p	An integer. For "savitzky-golay", indicates the polynomial order and must satisfy p < w and p >= m. For "gap-segment" and "nwp", indicates the segment or smoothing window size and must be a positive odd integer.

`algorithm` A character string specifying the algorithm. One of `"savitzky-golay"` (default), `"gap-segment"`, or `"nwp"`. See Details.

Details

Savitzky-Golay (`"savitzky-golay"`): fits a polynomial of order `p` within a moving window of size `w` and differentiates analytically. Implemented via `savitzkyGolay`.

Gap-Segment (`"gap-segment"`): computes the derivative over a gap of `w` points, with optional averaging over a segment of `p` points. When `p = 1` this reduces to the standard Norris-Gap derivative. Implemented via `gapDer`.

NWP (`"nwp"`): reproduces the "DG" derivative pre-treatment from BUCHI NIRWise PLUS calibration software. A moving average of window `p` is applied first (pre-smoothing), followed by differentiation. For first order, a gap derivative with gap `w` is used. For second order, a centered second difference with spacing `half_w` is computed:

$$d^2x_i = \frac{2x_i - (x_{i+h} + x_{i-h})}{2h}$$

where $h = half_w$. Edge columns affected by the window are removed from the output.

For the `"nwp"` algorithm, the NIRWise PLUS half-window conventions are:

$$half_w = (w + 1)/2$$

$$half_s = (p - 1)/2$$

These are stored internally for device file serialization and are not user-facing parameters.

Value

An object of class `preprocessing` to be used in `preprocess_recipe` and executed by `process`. The object is a list containing the method name, all parameters, and (for `algorithm = "nwp"`) the NIRWise PLUS half-window values (`half_w`, `half_s`) required for device file serialization.

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

`preprocess_recipe`, `process`

Examples

```
data("NIRcannabis")
X <- NIRcannabis$spc

# Savitzky-Golay first derivative, window 11, polynomial order 3
sg <- prep_derivative(m = 1, w = 11, p = 3, algorithm = "savitzky-golay")

# Gap-Segment second derivative, gap 9, segment 3
gs <- prep_derivative(m = 2, w = 9, p = 3, algorithm = "gap-segment")
```

```
# NWP first derivative, window 5, pre-smoothing 11
nwp <- prep_derivative(m = 1, w = 5, p = 11, algorithm = "nwp")

# Apply via preprocess_recipe
recipe <- preprocess_recipe(sg, device = "unspecified")
X_der <- process(X, recipe)
```

`prep_detrend` *Detrending constructor for spectral preprocessing*

Description

Creates a preprocessing constructor for detrending spectral data. The constructor is intended to be passed to `preprocess_recipe` and executed via `process`.

Usage

```
prep_detrend(p = 2)
```

Arguments

`p`	A positive integer specifying the polynomial order used for fitting. Must be >= 1. Default is 2.

Details

For each spectrum, a polynomial of order `p` is fitted using the column wavelengths as the explanatory variable (or integer indices if column names are not numeric). The residuals from this fit are returned as the detrended spectrum, removing wavelength-dependent baseline effects.

This constructor always performs pure polynomial detrending without a prior SNV transformation. Users who want the full Barnes et al. (1989) procedure (SNV followed by detrending) should chain `prep_snv` before `prep_detrend` in their recipe.

The computation is delegated to `detrend` with `snv = FALSE`.

Value

An object of class `preprocessing` to be used in `preprocess_recipe` and executed by `process`.

Author(s)

Leonardo Ramirez-Lopez

References

Barnes RJ, Dhanoa MS, Lister SJ. 1989. Standard normal variate transformation and de-trending of near-infrared diffuse reflectance spectra. Applied Spectroscopy, 43(5): 772-777.

See Also

prep_snv, preprocess_recipe, process

Examples

```
data("NIRcannabis")
X <- NIRcannabis$spc

# Pure polynomial detrend
dt <- prep_detrend(p = 2)
recipe <- preprocess_recipe(dt, device = "unspecified")
X_dt <- process(X, recipe)

# Barnes et al. (1989): SNV followed by detrend
recipe_barnes <- preprocess_recipe(
  prep_snv(), prep_detrend(p = 2), device = "unspecified"
)
X_barnes <- process(X, recipe_barnes)
```

---

prep_resample *Resampling constructor for spectral preprocessing*

Description

Creates a preprocessing constructor for resampling spectral data to a new wavelength grid. The constructor is intended to be passed to preprocess_recipe and executed via process.

Usage

```
prep_resample(grid)
```

Arguments

grid	Either a numeric vector of length 3 specifying the target wavelength grid as c(min_wav, max_wav, resolution), or the character string "proxiscout" to resample to the standard NeoSpectra wavenumber grid (see Details). When grid is a numeric vector: • grid[1] (min_wav): minimum wavelength of the target grid. • grid[2] (max_wav): maximum wavelength; must be greater than min_wav. • grid[3] (resolution): spacing between wavelengths; must be positive. Extrapolation beyond the range of the input wavelengths is never allowed.

Details

User-defined grid (`grid = c(min_wav, max_wav, resolution)`): resamples spectra to the specified target grid using natural spline interpolation via `resample`. Column names of `X` must be coercible to numeric wavelength values. This mode is compatible with the `"proximate"` device.

NeoSpectra grid (`grid = "proxiscout"`): resamples spectra to the standard wavenumber grid of NeoSpectra NIR scanners (approx. 3921.569 to 7407.407 cm^{-1}, ~256 channels at ~13.617 cm^{-1} steps). Only wavenumbers overlapping with the input range are retained. This mode is compatible with the `"proxiscout"` device.

Value

An object of class `preprocessing` to be used in `preprocess_recipe` and executed by `process`.

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

`preprocess_recipe`, `process`, `get_proxiscout_wavenumbers`

Examples

```
data("NIRcannabis")
X <- NIRcannabis$spc

# User-defined grid (proximate)
rs <- prep_resample(grid = c(1001, 1700, 2))
recipe <- preprocess_recipe(rs, device = "proximate")
X_rs <- process(X, recipe)
```

---

prep_smooth *Smoothing constructor for spectral preprocessing*

Description

Creates a preprocessing constructor for smoothing spectral data. The constructor is intended to be passed to `preprocess_recipe` and executed via `process`.

Two algorithms are supported: Savitzky-Golay (`"savitzky-golay"`) and moving average (`"moving-average"`).

Usage

```
prep_smooth(w, p = NULL, algorithm = c("savitzky-golay", "moving-average"))
```

Arguments

w	A positive odd integer specifying the filter window size.
p	An integer specifying the polynomial order. Required when algorithm = "savitzky-golay". Must satisfy p < w and p >= 0. Ignored for "moving-average".
algorithm	A character string specifying the smoothing algorithm. One of "savitzky-golay" (default) or "moving-average". See Details.

Details

Savitzky-Golay ("savitzky-golay"): fits a polynomial of order p within a moving window of size w and returns the zero-order coefficient (i.e. the smoothed value). Implemented via savitzkyGolay with m = 0.

Moving average ("moving-average"): computes a simple moving average of window size w using movav. Edge values are handled using progressively narrower windows so the output has the same number of columns as the input. This reproduces the "Smooth" pre-treatment from BUCHI NIRWise PLUS.

For "moving-average", the NIRWise PLUS half-window convention is:

$$half_w = (w - 1)/2$$

stored internally for device file serialization and not user-facing.

Value

An object of class preprocessing to be used in preprocess_recipe and executed by process. The object is a list containing the method name and all parameters. For algorithm = "moving-average", the NIRWise PLUS half-window value (half_w) is also stored for device file serialization.

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

preprocess_recipe, process

Examples

```
data("NIRcannabis")
X <- NIRcannabis$spc

# Savitzky-Golay smoothing, window 11, polynomial order 3
sg <- prep_smooth(w = 11, p = 3, algorithm = "savitzky-golay")

# Moving average smoothing, window 7
ma <- prep_smooth(w = 7, algorithm = "moving-average")

# Apply via preprocess_recipe
recipe <- preprocess_recipe(sg, device = "proxiscout")
```

```
X_smooth <- process(X, recipe)
```

`prep_snv` *Standard Normal Variate constructor for spectral preprocessing*

Description

Creates a preprocessing constructor for applying Standard Normal Variate (SNV) normalisation to spectral data. The constructor is intended to be passed to `preprocess_recipe` and executed via `process`.

Usage

```
prep_snv()
```

Details

SNV normalises each spectrum row-wise by subtracting its mean and dividing by its standard deviation:

$$SNV_i = \frac{x_i - \bar{x}_i}{s_i}$$

where x_i is the signal of the ith observation, $\bar{x}_i$ is its mean and s_i its standard deviation. Implemented via `standardNormalVariate`.

Value

An object of class `preprocessing` to be used in `preprocess_recipe` and executed by `process`.

Author(s)

Leonardo Ramirez-Lopez with code from Antoine Stevens

References

Barnes RJ, Dhanoa MS, Lister SJ. 1989. Standard normal variate transformation and de-trending of near-infrared diffuse reflectance spectra. Applied spectroscopy, 43(5): 772-777.

See Also

`preprocess_recipe`, `process`

Examples

```
data("NIRcannabis")
X <- NIRcannabis$spc

snv <- prep_snv()
recipe <- preprocess_recipe(snv)
X_snv <- process(X, recipe)
```

`prep_transform` *Reflectance/absorbance conversion constructor for spectral preprocessing*

Description

Creates a preprocessing constructor for converting spectral data between reflectance and absorbance. The constructor is intended to be passed to `preprocess_recipe` and executed via `process`.

Usage

```
prep_transform(to = c("absorbance", "reflectance"))
```

Arguments

`to`	A character string specifying the target unit. Either `"absorbance"` (default) or `"reflectance"`.

Details

Conversion follows Beer's Law:

$$A = -\log_{10}(R)$$

where A is absorbance and R is reflectance.

When converting to absorbance, all values in `X` must be strictly positive. A warning is issued if the resulting absorbance contains small negative values, which may indicate precision or scaling issues in the input.

Note that no check is performed on whether the input is actually in the expected unit (the transformation is applied as specified).

Value

An object of class `preprocessing` to be used in `preprocess_recipe` and executed by `process`.

Author(s)

Leonardo Ramirez-Lopez

See Also

preprocess_recipe, process

Examples

```
data("NIRcannabis")
X <- NIRcannabis$spc # absorbance

tr <- prep_transform(to = "reflectance")
recipe <- preprocess_recipe(tr, device = "proxiscout")
X_ref <- process(X, recipe)
```

---

prep_wav_trim *Wavelength trimming constructor for spectral preprocessing*

Description

Creates a preprocessing constructor for trimming spectral data to a specified wavelength band. The constructor is intended to be passed to preprocess_recipe and executed via process.

Usage

```
prep_wav_trim(
  band,
  trim_constant_edges = FALSE
)
```

Arguments

band	A numeric vector of length 2 giving the minimum and maximum wavenumber/wavelength to retain. Columns of X outside this range are dropped. Pass c() (empty vector) to skip band trimming and only apply trim_constant_edges.
trim_constant_edges	A logical. If TRUE, constant or zero-valued columns at the left and right edges are removed after band trimming. Default is FALSE.

Details

Band trimming retains only those columns whose names (coerced to numeric) fall within [min(band), max(band)]. If no columns fall within the band the original matrix is returned with a warning.

Constant edge trimming scans inward from each edge and drops columns that are identical to their immediate neighbour or are all zero. If trimming would leave fewer than two columns the step is skipped with a warning.

Value

An object of class preprocessing to be used in preprocess_recipe and executed by process.

Author(s)

Claudio Orellano and Leonardo Ramirez-Lopez

See Also

preprocess_recipe, process

Examples

```
data("NIRcannabis")
X <- NIRcannabis$spc

tr <- prep_wav_trim(band = c(1000, 1800))
recipe <- preprocess_recipe(tr, device = "proxiscout")
X_trim <- process(X, recipe)
```

---

proximate_add2nax *Prepare data for augmenting a nax application*

Description

This function collects all the necessary data that is required prior updating a nax application.

Usage

```
proximate_add2nax(formulas = NULL, data, metadata_list = NULL, skip_indices_list = NULL)
```

Arguments

formulas	a list containing one or more objects of class formula where each of them represents the model to be calibrated.
data	a data.frame containing the data of the variables in the model (as in the calibrate function).
metadata_list	a list of containing the specifications for the metadata of each model in formulas given in the same order. Each element in the list should be defined as in the metadata argument of calibrate using the add_model_metadata function. Defaults to NULL.
skip_indices_list	a list of vectors of integers for the indices in the input data to be skipped for the computation of each of the models in formulas. The vectors in this list must be provided in the same order as their corresponding counterparts in formulas. Defaults to NULL. In case a list is passed, the list components must be filled with numeric() for those formulas where there is no indices to be skipped.

Value

A list mirroing the objects passed to the function.

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

proximate_recalibrate_nax

calibrate,

preprocess_recipe,

fit_plsr,

fit_xlsr,

calibration_control,

calibrate_models

---

proximate_data *Create a data frame for NIRWise PLUS applications*

Description

Create a data frame of class "proximate_data", similar to proximate_read_data, but without the need for a file. Instead, data can be supplied directly from R.

Usage

```
proximate_data(
  spc, id, properties = NULL, row = seq_len(nrow(spc)), check = "True", date = Sys.time(),
   snr = NULL, barcode = "", note = "", begin = Sys.time(), end = Sys.time(),
   recipe = "", coeffs = NULL
)
```

Arguments

spc	A matrix containing the spectral data. Note that the names of the columns must indicate the corresponding wavelength range at which the spectra was measured. Hence, the column names must be convertible to numerical values.
id	A vector of length equal to the number of rows of spc. Corresponds to the ID of the spectra, and must be provided.
properties	Either NULL (default) or a matrix containing numerical values, indicating the reference values of each property, where the column names correspond to the names of the properties. If a matrix is provided, it must contain the same number of rows as spc, but can contain NA values.
row	A vector of length equal to the number of rows of spc. Contains the row number of the observation.

check	A vector of characters with length equal to the number of rows of spc or a single character. Must only contain characters "True" or "False". Defaults to "True".
date	A vector of length equal to the number of rows of spc or a single character. Indicates the date when the measurement was taken. Format should be: "year-month-day hour:min:sec". In case an object inheriting from "POSIXct", formatting will be done automatically. Defaults to Sys.time().
snr	A vector of length equal to the number of rows of spc, a single character or NULL. Indicated the serial number of the instrument the measurement was taken with. Defaults to NULL, in which case the serial numbers are all equal to "0000000000".
barcode	A vector of length equal to the number of rows of spc or a single character. Contains the barcodes for the measurement. Defaults to an empty string.
note	A vector of length equal to the number of rows of spc or a single character. Contains notes for the measurements. Defaults to an empty string.
begin	A vector of length equal to the number of rows of spc or a single character. Contains the time and date of the beginning of the measurement process. Format should be: "year-month-day hour:min:sec". In case an object inheriting from "POSIXct", formatting will be done automatically. Defaults to the system's current date and time.
end	A vector of length equal to the number of rows of spc or a single character. Contains the time and date of the ending of the measurement process. Format should be: "year-month-day hour:min:sec". In case an object inheriting from "POSIXct", formatting will be done automatically. Defaults to the system's current date and time.
recipe	A vector of length equal to the number of rows of spc or a single character. Contains the recipe for the measurements. Defaults to an empty string.
coeffs	A list with exactly three entries. Parameter is ignored if the wavelength resolution of spc is constant. For non-constant resolution, this parameter must be supplied. See details on this parameter needs to be defined. Default is NULL.

Details

This function provides an alternative way of creating a data.frame with the necessary structure that is required by many functions of this package. In particular, this function does not require any already existing files like proximate_read_data.

Note that only the first two arguments to this function are required for creating the data frame. However, the properties argument should most often also be provided, as these contain the necessary reference values for the process of modeling and creating an application with the spectral data.

Most parameters of this function can either have length equal to the number of rows of spc or length equal to one. In latter case, the value is recycled for every row of the returned data frame.

Furthermore, we emphasize that the column names of matrix spc must contain the wavelength ranges of the spectra.

In case these spectra do not have a constant resolution, the function will require additional information on how the spectral wavelength range can be recovered. Then, the parameter coeffs will be mandatory and must contain information on the polynomial coefficients that were used to obtain

the wavelengths. More information, including an example, can be seen in the vignette about the `vignette(ProxiMate-Structure-of-the-application-files)`. A concrete example is also given below.

The `coeffs` must be a named list with exactly 3 entries: `X1`, `X2`, `X3`. In ProxiMate data files (.tsv), they can be seen at columns #X1, #X2, #X3. Note that both `X1` and `X2` must be vectors of either length 1 or 2, containing the start and end pixels respectively, while `X3` is a list of length 1 or 2, containing polynomial coefficients as vectors of arbitrary length. The entries of the `coeffs` can either be for a near-infrared only (i.e. length 1), or for both the visible and near-infrared range (i.e. length 2).

The coefficients are attached to the returned `data.frame` as an attribute `"coeffs"`.

Value

A `data.frame` of class `proximate_data` containing all the metadata, response variables and spectra. The spectra is returned in a matrix embedded in the data.frame which can be accessed as `...$spc`.

Author(s)

Claudio Orellano

Examples

```
data("NIRcannabis")
dat <- NIRcannabis

# Reconstruct NIRcannabis with properties in a different order
spc <- dat$spc
properties <- matrix(
  c(dat$CBD, dat$CBDA, dat$THC, dat$THCA),
  ncol = 4, dimnames = list(NULL, c("CBD", "CBDA", "THC", "THCA"))
)
datc <- proximate_data(
  spc, dat$ID, properties, dat$ROW,
  date = dat$Date, snr = dat$SNR, barcode = dat$Barcode,
  note = dat$Note, begin = dat$Begin, end = dat$End, recipe = dat$Recipe
)

# They are similar to each other (except the order of properties):
dat_refs <- which(names(dat) %in% c("Reference", colnames(properties)))
datc_refs <- which(names(datc) %in% c("Reference", colnames(properties)))
all.equal(dat[, -dat_refs], datc[, -datc_refs]) # TRUE

# In case of non-constant wavelengths, have to pass the coefficients to the function.
# Coefficients are usually given as #X1, #X2, #X3 in ProxiMate .tsv files,
# e.g. using coefficients example of vignette(Structure-of-the-application-files):
coeffs <- list(
  X1 = c(823, 4),
  X2 = c(1074, 272),
  X3 = list(
    c(0, 0, 0, -3.618926e-05, 2.137782, -1.333363e+03),
```

```
    c(2.04E-10, -1.28E-07, 2.80E-05, -4.76e-3, 3.89, 880.06)
  )
)

# You can extract the wavelengths in nm using these coefficients like this:
# Note that NIR pixels must be shifted by one to the right, as they are zero-based
pixel_seq <- list((coeffs$X1[1]:coeffs$X2[1]), (coeffs$X1[2]:coeffs$X2[2]) + 1)
vis_wavs <- mapply(
  pixel_seq[[1]],
  FUN = function(x) coeffs$X3[[1]] %*% c(x^5, x^4, x^3, x^2, x^1, 1)
)
nir_wavs <- mapply(
  pixel_seq[[2]],
  FUN = function(x) coeffs$X3[[2]] %*% c(x^5, x^4, x^3, x^2, x^1, 1)
)
wavs <- c(vis_wavs, nir_wavs)

# Above coefficients now have to be passed to the proximate_data()
# function since there are non-constant wavelengths.

# If we (wrongly) assume that NIRcannabis has such wavelengths:
rand_mat <- matrix(rnorm((length(wavs) - ncol(spc)) * nrow(spc)), nrow = nrow(spc))
spc <- cbind(rand_mat, spc)
colnames(spc) <- wavs

# Now we can create data object with coefficients
datcc <- proximate_data(
  spc, dat$ID, properties, dat$ROW,
  date = dat$Date, snr = dat$SNR, barcode = dat$Barcode,
  note = dat$Note, begin = dat$Begin, end = dat$End, recipe = dat$Recipe,
  coeffs = coeffs
)

# Coefficients can be viewed with
attr(datcc, "coeffs")
```

`proximate_merge` *Merge datasets of class* `proximate_data`

Description

This function allows you to quickly merge two separate datasets of class `proximate_data` into a single one. The first dataset must be of class `proximate_data`, while the second may be any kind of list-like format, but must contain at least columns named `spc` and `ID`.

Usage

```
proximate_merge(x)
```

Arguments

x	a list containing objects of class proximate_data, obtained from proximate_read_data or via proximate_data. The first element in the list is used as the reference for aligning the spectral data of the remaining elements. See details.

Details

This functions provides a way to merge different datasets into a single table.

In cases where the first dataset in the list (the one used as reference for spectral alignment) has spectral data with an spectral range outside the limits of another dataset, the spectral data of such dataset will not be extrapolated. In that case the spectral variables outside such limits will be filled with NAs.

The function checks for any of the standard names of a .tsv file of ProxiMate, identifying any unexpected column names as properties.

Propeties that are contained in both datasets are merged into a single column. Otherwise, the columns of a property that is only contained in one of the datasets is filled up with NA.

Value

a data.frame of class proximate_data, containing the merged data.

Author(s)

Claudio Orellano

See Also

proximate_read_data, proximate_data

Examples

```
# to do
```

---

proximate_read_cal *Read model parameters from ProxiMate .cal files*

Description

Reads the metadata and model parameters from one or more .cal files generated by BUCHI ProxiMate sensors. The function extracts the preprocessing recipe, regression method, PLS weights, loadings, scores, intercepts, and bias terms required to project new spectra into the score space and produce predictions. Spectral regression coefficients are not retrieved directly; predictions are computed in the score space via predict.read_cal.

Usage

```
proximate_read_cal(file, ignore_version = FALSE)

## S3 method for class 'read_cal'
predict(object, newdata, get_comp = c("optimal", "all"),
        get_scores = FALSE, bias_index = 1, ...)
```

Arguments

file	a character vector of .cal file paths.
ignore_version	a logical. If FALSE (default), files with no version information or created with NIRWise PLUS prior to version 1.0 raise an error. If TRUE, such files are read with a warning instead; predictions from these files may deviate from those produced on the instrument.
object	an object of class read_cal as returned by proximate_read_cal().
newdata	a matrix of new spectral data to predict from. Column names must be coercible to the wavelengths used in the model.
get_comp	a character string. Either "optimal" (default) to return predictions only for the optimal number of components, or "all" to return predictions for every available component.
get_scores	a logical indicating whether PLS scores should be returned alongside predictions. Default is FALSE.
bias_index	the index of the bias to be applied in the list of biases. These are generated in NIRWise PLUS based on the number of files containing the calibration data. Default = 1.
...	not currently used.

Value

For proximate_read_cal(), a list of class "read_cal" with the following elements:

- summary: a data.frame describing each model:
  - Property: name of the response variable.
  - Preprocessing: sequence of preprocessing steps applied (without parameters).
  - Method: regression method used.
  - Factors: number of PLS components used.
  - Cross-validation: number of cross-validation segments. A value of 0 indicates no cross-validation was used.
  - Auto-skip: logical indicating whether automatic outlier removal (auto-delete) was applied during calibration.
- meta_param: a list with one element per model containing the preprocessing recipe (precipe), the indices of automatically removed observations (auto_skip), and a logical indicating whether sample aggregation was applied (aggregate).
- file_info: a list with one element per model containing the file paths of the spectral data used for calibration (files) and the indices of manually skipped observations per file (skipped_indices).

- `models`: a list with one element per model containing all parameters required for prediction: wavelengths, preprocessing recipe, number of factors, mean-centering vector, scores, score scale factors, PLS weights, loadings, biases, intercept, and target values.

For `predict.read_cal()`, a list with the following elements:

- `predictions`: predicted values for each model in `object`.
- `distances`: scaled score distances for each sample and model, which can be used to assess how well a new sample is represented by the model.
- `scores`: only returned when `get_scores = TRUE`. The projection of new samples into the PLS score space.

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

`proximate_recalibrate_nax`, `proximate_read_nax`

`proximate_read_data` *Read ProxiMate (.tsv) files*

Description

This function imports .tsv files generated by BUCHI ProxiMate sensors.

Usage

```
proximate_read_data(file)
```

Arguments

`file`	A string indicating the name (and path) of the .tsv file. A `textConnection` can also be passed.

Value

A data.frame containing all the metadata, response variables and spectra in the tsv file. The spectra is returned in a matrix embedded in the data.frame which can be accessed as `...$spc`.

Author(s)

Leonardo Ramirez-Lopez

Examples

```
data("NIRcannabis")
filename <- paste0(tempdir(), "/NIRcannabis.tsv")
# Need to produce a tsv file before we can read it
proximate_write_data(
  x = NIRcannabis,
  file = filename,
  properties = c("CBDA", "THCA", "CBD", "THC")
)
# Equivalent to dataset NIRcannabis
dat <- proximate_read_data(filename)
```

`proximate_read_nax` *Reads and summarizes ProxiMate spectroscopic applications (nax files)*

Description

This function reads and summarizes the main aspects of BUCHI ProxiMate applications which are files of extension .nax. In addition, the file is retain as raw binary in the object generated by this function.

Usage

```
proximate_read_nax(file, ignore_version = FALSE)
```

Arguments

`file`	a character vector containing the .nax file name (and path).
`ignore_version`	a logical passed to `proximate_read_cal`. If `FALSE` (default), embedded `.cal` files with no version information or created with NIRWise PLUS prior to version 1.0 raise an error. Set to `TRUE` to read them with a warning instead.

Value

A a list of class `nax` which contains the following objects:

- `nax_summary`: a list with:
  - `content`: the name of the files inside the nax/application.
  - `size`: the size (on disk) of the nax.
  - `raw`: the original nax file/application stored as raw binary.
- `nad_info`: a list with:
  - `summary`: a summary of the high-level ProxiMate application parameters.
  - `data`: a full list of the high-level ProxiMate application parameters.
- `cal_info`: a list with:
  - `summary`: a summary of the calibration models contained in the ProxiMate application.

  - meta_param: a list with parameters of each calibration model (e.g. pre-processing recipes.
- rtf_info: a list with:
  - summary: a summary of the calibration models as printed in the calibration reports contained in the nax file. This includes the optimal number of components suggested (ncomp).
- data: a list with:
  - summary: a summary of the calibration data (tsv files) contained in the ProxiMate application.
  - data: a list with the calibration data found in all the tsv files.

In case, any of the above components is encrypted a character string indicating so will be returned. In case of the rtf calibration reports are not present in the nax, a NULL will be returned for rtf_info.

Author(s)

Leonardo Ramirez-Lopez

`proximate_recalibrate_nax` *Recalibrate a nax file*

Description

This function updates a nax file

Usage

```
proximate_recalibrate_nax(x,
                preprocess_recipes = NULL,
                methods = NULL,
                control = calibration_control(seed = 1),
                name,
                add = NULL)
```

Arguments

x	an object of class nax as returned by the proximate_read_nax function.
preprocess_recipes	an optional list with one or more objects of class preprocess_recipe that are to be tested for finding the optimal one for each model in the list passed to formulas.
methods	an optional list containing one ore more objects of class fit_constructor which are as returned by one of the fit_constructors functions, indicating what type of regression method to use along with its parameters.

control	a calibration_control object as returned by the calibration_control function, indicating how some aspects of the calibration process must be conducted (e.g. cross-validation and outlier detection). Default is calibration_control(seed = 1). See details.
name	a vector length at most 2, consisting of characters for the name and alias of the application. Defaults to "Untitled".
add	an optional object of class nax_augment as returned by the proximate_add2nax function.

Value

A list of class "spectral_multimodel". See calibrate_models function.

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

proximate_add2nax

calibrate,

preprocess_recipe,

fit_plsr,

fit_xlsr,

calibration_control,

calibrate_models

---

proximate_write_data *Write NIRWise PLUS readable tab-separated files*

Description

This function writes tab-separated value files in a readable NIRWise PLUS software format. These files contain visible and Near-Infrared absorbance spectra along with response variables and metainformation (e.g. sample ID, date, comments, etc).

Usage

```
proximate_write_data(x, file, id, spc, spc_round = 8, barcode = "", properties = NULL,
          note = "", recipe = "", created, snr)
```

Arguments

x	a data.frame of spectral data and metadata, for which the tab separated value file should be generated. See details.
file	a character for the path (and name) in which the tsv will be saved.
id	a vector of characters of length equal to the number of observations in x or of length 1. Each entry gives an observation a specific ID. If length is 1, this entry is recycled for every observation.
spc	either a character or a vector of integers. Specifies where the spectra can be found inside x. Default is "spc".
spc_round	an integer. To how many decimal places should the spectra be rounded? Defaults to 8 decimal places.
barcode	a vector of characters of length equal to the number of observations in x or of length 1. Each entry specifies the barcode for each observation; if length is 1, the entry is recycled for every observation. Default is an empty character.
properties	a vector of characters of arbitrary length. Which properties in x are to be added to the tsv? Note that any missing reference values for the properties are set to 0. Default is NULL.
note	a vector of characters of length equal to the number of observations in x or of length 1. The vector corresponds to the notes for each observation, or, if the vector is of length one, to all notes of all observations. Defaults to an empty character.
recipe	a vector of characters of length equal to the number of observations in x or of length 1. This vector corresponds to the recipe for each observation, or, if the vector is of length one, to all recipes of all observations. Defaults to an empty character.
created	a vector of characters of length equal to the number of observations in x. This vector should contain the date and time of when each observation was measured. If not provided and not contained in x, this parameter will be set to the current date and time of the system.
snr	a vector of characters, corresponding to the serial number of the device on which the measurement was taken. If not provided and not found in x, this parameter will be set to a vector of character of length equal to the number of rows in x, where each individual character is given by 0000000000.

Details

This function creates a tab separated value file, which is readable by both NIRWise PLUS software and the proximate_read_data function.

The main usage is to transform an already given data file into a format which is readable by NIRWise PLUS. Therefore, if some data of the given object x is already of the correct form, one can pass the corresponding values simply by passing the specific row of x to this function; for example, by passing note = x$Note.

Value

Invisibly returns NULL. Called for its side effect of writing a tab-separated value file to file.

Author(s)

Leonardo Ramirez-Lopez

Examples

```
data("NIRcannabis")
filename <- file.path(tempdir(), "NIRcannabis.tsv")

proximate_write_data(
  x = NIRcannabis,
  file = filename,
  id = NIRcannabis$ID,
  spc = "spc",
  spc_round = 8,
  barcode = NIRcannabis$Barcode,
  properties = c("CBDA", "THCA", "CBD", "THC"),
  note = NIRcannabis$Note,
  recipe = NIRcannabis$Recipe,
  created = NIRcannabis$Begin
)

# Since we do not change anything, the following produces the same tsv:
proximate_write_data(
  x = NIRcannabis,
  file = filename,
  properties = c("CBDA", "THCA", "CBD", "THC")
)
# Delete the file
file.remove(filename)
```

---

proximate_write_model *Write calibration (.cal), project (.prj) and report (.rtf) files to a specified directory*

Description

This function allows to write native ProxiMate calibration, project and report files from a `spectral_model` object.

Usage

```
proximate_write_model(object, path, tsv_paths, application_name = "Untitled",
                      cal = TRUE, prj = TRUE, rtf = TRUE,
                      verbose = TRUE, internal_prj_path = NULL)
```

Arguments

object	a list of models of class spectral_model. These models should be generated using the calibrate function.
path	a string for the directory in which the files should be saved.
tsv_paths	a vector of character strings for the paths (including the names) of the tsv data files. See details.
application_name	a string with the name of the generated files. Defaults to "Untitled".
cal	a logical. Should a calibration file (.cal) be written? Default is TRUE.
prj	a logical. Should a project file (.prj) be written? Default is TRUE.
rtf	a logical. Should a report in rich text format (.rtf) be written? Default is TRUE.
verbose	a logical. Should progress bars for the generated files be printed? Default is TRUE.
internal_prj_path	a string. Only used for changing the path printed on the first line of the project file. This is necessary mainly for calls from proximate_write_nax, as it creates the project file in a temporary file, which would also store that temporary path into the project file. This argument allows you to overwrite that path individually. Otherwise, this parameter may be ignored. If NULL (default), will be set to path.

Details

This function generates files with extensions ".prj" (project file), ".cal" (calibration file), and ".rtf" (report) for the provided models of class spectral_model in the argument object. Each file type can be individually enabled or disabled via the cal, prj, and rtf arguments. All files will be named according to the chosen name of the application (given by application_name). Note that in contrast to proximate_write_nax, the metadata does not influence the name of the application. This allows models to be passed directly to this function without the need for metadata. Additionally, the name of the response variable is automatically added to the names of the produced files, so that all generated files have unique names.

Value

Invisibly returns NULL. Called for its side effect of writing calibration, project and/or report files to path.

Author(s)

Claudio Orellano, Leonardo Ramirez-Lopez

Examples

```
data("NIRcannabis")
control <- calibration_control(validation_type = "kfold", number = 3, folds = "sequential")
amodel <- calibrate(CBDA ~ spc,
  data = NIRcannabis, preprocess = preprocess_recipe(),
```

```
  method = fit_plsr(5), control = control, verbose = FALSE
)

proximate_write_model(
  object = list(amodel),
  path = tempdir(),
  tsv_paths = tempfile(fileext = ".tsv"),
  application_name = "Untitled",
  cal = TRUE, prj = TRUE, rtf = TRUE,
  verbose = FALSE
)
```

`proximate_write_nax` *Create an application file for the given list of models*

Description

This function provides a flexible way to create an application file (.nax) which can be deployed into ProxiMate sensors.

Usage

```
proximate_write_nax(
   object, path, metadata, tsv_name, empty_tsv_name, spc = "spc",
   external_properties = NULL, report = TRUE, verbose = TRUE,
   internal_prj_path = NULL
)
```

Arguments

`object`	a list of objects of class `spectral_model`, as generated by `calibrate`. Note that at least one of these models must contain the relevant input data (i.e. run with `return_inputs = TRUE` in `calibrate` method).
`path`	a character for the directory in which the file should be produced.
`metadata`	an object of class `application_metadata`, as generated by `add_application_metadata`. If not provided (and `object` does not contain the metadata either), default values are used with a warning.
`tsv_name`	an optional character. If not supplied, this parameter is set to the name of the application plus the current date. See details.
`empty_tsv_name`	an optional character. For ProxiMate applications, this argument should be different to `tsv_name` or it cannot be imported into the device. If missing, is set to the name of the application. See details.
`spc`	a character to indicate the column name of the spectra used in the data provided to the `object` list of models. This parameter is passed to the `proximate_write_data` method. Defaults to `"spc"`.

external_properties
: a list for additional files to be included in the application file. Defaults to NULL. See details for how these files may be added.

report
: a logical. Should reports of the models be generated and added to the file? Defaults to TRUE.

verbose
: a logical indicating whether progress bars during the creation of the file should be printed. Defaults to TRUE.

internal_prj_path
: a string. Only used for changing the path printed on the first line of each project file. For almost all cases, this argument can be ignored. The only case where you should adjust this parameter is when you are creating the application (.nax) file in a certain folder, but actually want to move it to another one (e.g. on a different platform). If NULL (default), the path is set equal to path, with "Calibrations/" added at the end.

Details

This function is capable of generating an application (.nax) file, which contains compressed data files for the application. All files inside this .nax file are organized in a fixed way, such that they are importable into a ProxiMate device. For that, all models to be imported should be in a list, and each individual model should be generated using the calibrate function, preferably with the input data saved in it. This can easily be done by calling the method with return_inputs = TRUE. Note that at least one model in object must contain input data, otherwise an error will occur.

Furthermore, note that the data argument in calibrate for all models in one single application must be from one single data set. In particular, one single data.frame must suffice to describe the inputs of all models in object. The data that is actually used to train these models can still be different, e.g. by specifying the rows that you want to exclude from a certain model (see skip argument of calibrate). An error will be thrown if this is not the case.

The directory path is created automatically (if it does not exist). Inside, the application file is generated, which contains the following compressed files: a file for the metadata (.nad), project (.prj) and calibration (.cal) files for all the provided models in object, possibly report (.rtf) files (as indicated by the report argument), a tab-separated value (.tsv) file of the spectral data, and an empty tab-separated value (.tsv) file.

The metadata file (.nad) is required for a successful import of the application into a ProxiMate device. This requires metadata in every model, which should be added using add_model_metadata prior to the call of this function. Otherwise, default values for the model metadata will be used with a warning. Furthermore, application specific metadata is required, which can be either specified by providing the argument metadata, or included in the list of models object (see add_application_metadata), where the former option will take precedence. If neither option is available, default values of add_application_metadata are used with a warning.

Furthermore, this function provides a way of adding separately generated project and calibration files through the parameter external_properties. Note that these files have to be either in the directory of the provided path or in a sub-directory "Calibrations" thereof. External properties must be provided as a list containing model metadata (using the add_model_metadata method) in order to be added properly to the application file.

These external files must also be named according to the naming convention of the rest of the models used. In particular, the function searches the provided path and the sub-directory "Calibrations" for

files named with the following format: `app_name.property_name.cal`, `app_name.property_name.prj` and (if report is `TRUE`) `app_name.property_name.rtf`, where the `app_name` is taken from the application metadata, and the `property_name` from model metadata passed to `external_properties`. If the files cannot be found, a warning will be displayed.

An example for adding an external property is given in the example section below.

Note that if an application file for the given application already exists, the files inside the compressed application file are updated, but already present files are not deleted.

Value

Invisibly returns `NULL`. Called for its side effect of writing a `.nax` application file to `path`.

Author(s)

Claudio Orellano, Leonardo Ramirez-Lopez

Examples

```
data("NIRcannabis")
control <- calibration_control(validation_type = "kfold", number = 3, folds = "sequential")
# Models for application files must have model metadata!
model_metadata <- add_model_metadata(unit = "%")
modell <- calibrate(CBDA ~ spc,
  data = NIRcannabis, preprocess = preprocess_recipe(),
  method = fit_plsr(15), control = control,
  metadata = model_metadata, verbose = FALSE
)

app_metadata <- add_application_metadata(name = "app")
proximate_write_nax(
  object = list(modell),
  path = tempdir(),
  metadata = app_metadata,
  tsv_name = "some_tsv",
  empty_tsv_name = "another_tsv",
  report = TRUE,
  verbose = FALSE
)

# Another model
modelr <- calibrate(THCA ~ spc,
  data = NIRcannabis, preprocess = preprocess_recipe(),
  method = fit_plsr(15), control = control,
  metadata = model_metadata, verbose = FALSE
)

# Generate some files to be added separately
proximate_write_model(
  object = list(modelr),
  path = tempdir(),
  tsv_paths = tempdir(),
```

```
  application_name = "app",
  cal = TRUE, prj = TRUE, rtf = TRUE,
  verbose = FALSE
)

# Now add them using external properties. Requires a name for the property!
proximate_write_nax(
  object = list(modell),
  path = tempdir(),
  metadata = app_metadata,
  tsv_name = "some_tsv",
  empty_tsv_name = "another_tsv",
  external_properties = list(add_model_metadata(unit = "%", name = "THCA")),
  report = TRUE,
  verbose = FALSE
)
```

`proxiscout_read_data` *Read and parse ProxiScout data from CSV or XLSX files*

Description

Reads spectral data files in either `.csv` or `.xlsx` format, identifies spectral data columns based on numeric column names, converts reflectance values from percentages to absolute units, and stores them in a matrix under the `spc` column.

Usage

```
proxiscout_read_data(file, references_file)
```

Arguments

`file`	A character string specifying the path to the input file. The file must be either have `.csv` or a `.xlsx` extension.
`references_file`	An optional character string specifying the path to a file containing reference values. See details.

Details

This function allows the user to give the path to one or two files at once.

If two file paths are given, the files are assumed to contain the spectral data in `file`, while `references_file` contains only the reference values. Both files must have a column that contains the regex `sample`, and the entries must coincide (excluding potential repetition identificators). These files are then merged together by the column with the name containing `sample`.

If only `file` is given, it must contain the spectral columns, and may or may not contain reference values.

In general, inside `file`, any column AFTER the spectra are identified as predictions, and are collected into a `matrix` called `predictions` (if any exist). Columns that contain numerical values and do not contain typical column names (see `extract_property_names` for more details) that appear BEFORE the spectral data columns are identified reference values.

The function:

- ensures the file extensions are valid (`.csv` or `.xlsx`).
- reads CSV files using `read.csv()` and Excel files using `readxl::read_excel()`.
- extracts spectral data (columns with numeric names).
- if exactly 257 columns with numeric names are found, then:
  - the spectral matrix is assigned the typical proxiscout wavenumbers (`get_proxiscout_wavenumbers`)
  - the data is assigned class `"proxiscout_data"`
  - spectral matrix is converted from percentage (0 to 100) to absolute (0 to 1) units.
- if the number of columns with numeric names is not 257, the spectral matrix is assigned the wavelengths/wavenumbers in the header of the file.
- stores the spectral data in a matrix named `spc`.
- stores columns after the spectral data in a matrix named `predictions` (if any exist).
- merges files together by the sample column if multiple files are given.

Value

A `data.frame` where:

- Spectral data is stored as a **matrix** in the `spc` column.
- Columns identified as predictions are stored as a **matrix** in the `predictions` column.
- Other non-spectral metadata columns remain unchanged.
- Multiple files are merged into a single `data.frame`.
- If the files contain 257 columns in `spc`, the data is assigned class `"proxiscout_data"`.

Note

This function assumes spectral column names follow a strict numeric pattern (e.g. "3921.0") and removes any prefixed characters such as "X" that may be added by `read.csv`. These names are converted to numeric and used as column names of the spectral matrix.

Author(s)

Leonardo Ramirez-Lopez, Claudio Orellano

proxiscout_repetition_pattern *ProxiScout repetition pattern*

Description

Returns the pattern that can be used to identify repetitions in the sample ID of ProxiScout data files

Usage

```
proxiscout_repetition_pattern()
```

Value

A character that can be used as a regex for identifying repetitions in ProxiScout data files

Author(s)

Claudio Orellano

proxiscout_write_data *Write data files for ProxiScout devices*

Description

This function writes comma-separated files in a format compatible with ProxiScout-related software, which typically require two separate comma-separated files - one file for the spectra, and another file for reference values. These files are created inside the specified directory (argument `path`).

Usage

```
proxiscout_write_data(x, path, file_prefix, properties = NULL, spc = "spc")
```

Arguments

`x`	a `data.frame` of spectral data for which to write the data files. Typically, this is returned by `proxiscout_read_data` and of class `"proxiscout_data"`.
`path`	a character for the directory in which the files will be saved.
`file_prefix`	a character for the prefix of the generated files. The files are then named as `[file_prefix]_spectra.csv` and `[file_prefix]_properties.csv`. Default is `proxiscout_export`.
`properties`	a vector of characters of arbitrary length. Which properties in `x` are to be added to the csv? Default is `NULL`.
`spc`	either a character or a vector of integers. Specifies where the spectra can be found inside `x`. Default is `"spc"`.

Details

This function creates up to two comma separated files in the directory `path`, which are usable by ProxiScout-related software. These files are named according to the `file_prefix` argument and contain the spectra together with the sample names and device ID, respectively the reference values with the sample names.

Typically, the data provided to this function is imported with `proxiscout_read_data` and of class `"proxiscout_data"`, but it is also possible to construct a `data.frame` by hand and provide it to this function.

The `properties` argument specifies which columns in `x` are the reference values written to the `[file_prefix]_properties.csv` file. If empty (default), this file is not created, as it would only contain sample names. Any row in the provided properties that only contains `NA` values are dropped. In general, `NA` values are set to an empty string (`""`)

The sample names are detected automatically from `x` as the column with a name that contains `"sample"`. If none are detected, the function will throw an error. This column will be named `"Sample Name"` in the `[file_prefix]_spectra.csv` file, and `"sampleName"` in the `[file_prefix]_properties.csv` file.

Similarly, the device ID is a required column and is identified as having a `"device"` string inside the name of the column. This column is only written into the `[file_prefix]_spectra.csv` file, with a fixed named `"Device Id"`.

All other columns in either file only correspond to the spectra respectively the reference values. In particular, other columns in `x` are dropped.

Value

A `character` with the paths to the created files.

Author(s)

Leonardo Ramirez-Lopez, Claudio Orellano

`proxiscout_write_model` *Write a calibration model to ProxiScout JSON format*

Description

Serializes a model of class `spectral_model` (including its preprocessing recipe) into a JSON format that can be imported into the NeoSpectra NIR Hub and deployed on ProxiScout sensors (see Details).

Usage

```
proxiscout_write_model(object, file = NULL)
```

Arguments

`object`	an object of class `spectral_model` that contains the preprocessing recipe and final model to be serialized.
`file`	an optional character string with the path (including file name) where the JSON output should be written. If `NULL` (default), no file is written and the JSON string is returned. If a path is provided, the JSON is written to that file and returned invisibly.

Details

The JSON output produced by this function can be imported into the NeoSpectra NIR Hub and used within a ProxiScout application. Once imported, the NeoSpectra Scan mobile app linked to a ProxiScout sensor can access the model and use it to compute and display spectral predictions.

The JSON pipeline always begins with two hardware-specific steps that are added automatically, regardless of the preprocessing recipe in `object`: (1) scaling raw reflectance from the 0–100 range reported by the sensor to the 0–1 range, and (2) averaging repeated scans of the same sample. These steps precede any user-defined preprocessing.

Constraints and supported preprocessing steps:

- The first step in the preprocessing recipe of `object` must be `prep_resample`, as wavenumber alignment with the ProxiScout hardware grid is required.
- All predictor wavenumbers in `object` must match the hardware wavenumbers returned by `get_proxiscout_wavenumbers` within a tolerance of 0.1 cm^{-1}.
- `prep_derivative` and `prep_smooth` are supported only when `algorithm = "savitzky-golay"`.
- `prep_transform` is supported only with `to = "absorbance"`; using `to = "reflectance"` generates a warning and the step is skipped in the JSON output.
- `prep_wav_trim` is handled implicitly through wavenumber selection and does not produce an explicit JSON step.

Value

If `file = NULL` (default), the JSON string is returned visibly so it can be inspected or assigned to a variable. If `file` is specified, the JSON string is written to that file and returned invisibly (i.e. it is not printed to the console, following the standard R convention for functions called primarily for their side effect).

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

`calibrate`, `get_proxiscout_wavenumbers`, `prep_resample`

Examples

```
data("NIRcannabis")
control <- calibration_control(
  validation_type = "kfold", number = 3, folds = "sequential"
)
recipe <- preprocess_recipe(
  prep_resample(grid = "proxiscout"),
  prep_snv(),
  prep_derivative(m = 1, w = 11, p = 2, algorithm = "savitzky-golay"),
  device = "proxiscout"
)
model <- calibrate(
  THCA ~ spc,
  data = NIRcannabis, preprocess = recipe,
  method = fit_plsr(10), control = control, verbose = FALSE
)

json_model <- proxiscout_write_model(model)
json_model

proxiscout_write_model(model, file = file.path(tempdir(), "my_model.json"))
```

---

read_spc *Read and format spectral data from a file*

Description

This function reads spectral data from a file and extracts the spectral columns based on a specified prefix, or a range of columns. It can handle various delimiters and decimal separators.

Usage

```
read_spc(file, sep = "\t", dec = ".", header = TRUE, spectra_prefix = "",
         spectra_starts = NA, spectra_ends = NA, ...)
```

Arguments

file	a character string specifying the path to the file containing the spectral data.
sep	a character string indicating the field separator character. Defaults to "\t".
dec	a character string used for decimal points. Defaults to ".".
header	logical value indicating whether the file contains the names of the variables as its first line. Defaults to TRUE
spectra_prefix	a character string specifying the prefix used for spectral column names. If empty, the function will use column indices instead.
spectra_starts	an integer indicating the starting column index for the spectral data, used when spectra_prefix is not specified.

spectra_ends	an integer indicating the ending column index for the spectral data, used when spectra_prefix is not specified. If not provided, defaults to the last column.
...	additional arguments passed to read.table.

Details

The function reads a file and extracts the spectral data based on either a column name prefix or specified column indices. The spectral data is returned as a matrix in the spc column of the resulting data frame.

Value

a data frame with the original data and a matrix of spectral data stored in the spc column.

Author(s)

Leonardo Ramirez-Lopez

Examples

```
# write a file with spectra
data("NIRsoil", package = "prospectr")
spc_small <- NIRsoil$spc[1:5, ]
colnames(spc_small) <- paste0("X", colnames(spc_small))
tmp_df <- data.frame(ID = 1:5, Nt = NIRsoil$Nt[1:5], spc_small, check.names = FALSE)
tmp_file <- tempfile(fileext = ".txt")
write.table(tmp_df, file = tmp_file, sep = "\t", row.names = FALSE)

# read that
result <- read_spc(tmp_file, spectra_prefix = "X")
```

---

spectral_fit *The spectral_fit class*

Description

An object of class spectral_fit represents a fitted PLS or XLS regression model for a single component sequence. It is produced internally by calibrate and is accessible via object$final_model$model.

A spectral_fit object is a list with the following elements:

- method: The fit_constructor object passed to the fitting call. See fit_plsr and fit_xlsr.
- explained_variance: A list with two matrices: x_variance (three rows: pls_var, x_expl_var, x_expl_var_cum - absolute, relative, and cumulative relative explained variance of X per component) and y_variance (relative explained variance of the response per component).
- x_means: Named numeric vector of column means of the input spectral matrix X.

- `weights`: Matrix of PLS weights (one row per component).
- `scores`: Matrix of scores (one column per component).
- `sd_scores`: Named numeric vector of standard deviations for each score column.
- `scaled_scores`: Matrix of scores scaled by their standard deviations.
- `x_loadings`: Matrix of X loadings (one row per component).
- `projection_m`: Projection matrix that maps new spectra onto the score space.
- `intercept`: Named numeric scalar; the intercept of the regression model (equal to the mean of `Y`).
- `coefficients`: Matrix of regression coefficients (one row per component, one column per wavelength).
- `fitted_y`: Matrix of fitted response values (one column per component).
- `cal_error`: Matrix with three columns: number of components, root mean squared error of calibration, and largest residual.
- `x_residuals`: Matrix of spectral residuals (one column per component).
- `n_observations`: Integer; number of observations used for fitting.
- `y_quantiles`: Named numeric vector of the 0th, 25th, 50th, 75th, and 100th percentiles of the response `Y`.

Author(s)

Leonardo Ramirez-Lopez and Claudio Orellano

See Also

`calibrate`, `fit_plsr`, `fit_xlsr`

---

validate_prediction *Validate predictions of class* `'spectral_prediction'`

Description

Calculate several prediction validation statistics for a prediction of class `'spectral_prediction'`.

Usage

```
validate_prediction(prediction, reference)
```

Arguments

`prediction`	an object of class `'spectral_prediction'`, as returned by the `predict` function.
`reference`	a vector or a matrix with one column, containing the response variable.

Value

An object of class "spectral_validation", which is a list containing the following validation statistics of the prediction:

- model_information: A list containing information of the model on which the predictions are based. Mirrors the very same list contained in the prediction. See predict for more details.
- validation: A list with the validation statistics. For each prediction contained in prediction (which are based on the number of components), one entry in the list is added. Each of these elements exactly one matrix and one vector: val_results contains the predicted values and the corresponding errors in a matrix, while val_stats is a vector consisting of the coefficient of determination (R^2), root mean squared error (RMSE) and the largest residual obtained. These statistics are computed based on the prediction and reference, while ignoring any NA's.

Author(s)

Claudio Orellano

Examples

```
data("NIRcannabis")
skips <- c(10, 25, 37)
simple_model <- calibrate(CBDA ~ spc,
  data = NIRcannabis, preprocess = preprocess_recipe(),
  method = fit_plsr(5), control = calibration_control("kfold"),
  skips = skips, verbose = FALSE
)

# Predict the skipped indices
pred <- predict(simple_model,
  newdata = NIRcannabis[skips, ],
  ncomp = simple_model$final_ncomp,
  verbose = FALSE
)

# Validate skipped indices
validate_prediction(pred, NIRcannabis$CBDA[skips])
```

Index