Package ‘recipes’

July 8, 2022

Title Preprocessing and Feature Engineering Steps for Modeling

Version 1.0.1

Description A recipe prepares your data for modeling. We provide an extensible framework for pipeable sequences of feature engineering steps provides preprocessing tools to be applied to data. Statistical parameters for the steps can be estimated from an initial data set and then applied to other data sets. The resulting processed output can then be used as inputs for statistical or machine learning models.

License MIT + file LICENSE

URL `https://github.com/tidymodels/recipes`, `https://recipes.tidymodels.org/`

BugReports `https://github.com/tidymodels/recipes/issues`

Depends dplyr, R (>= 3.4)

Imports cli, ellipsis, generics (>= 0.1.2), glue, gower, hardhat (>= 1.2.0), ipred (>= 0.9-12), lifecycle, lubridate (>= 1.8.0), magrittr, Matrix, purrr (>= 0.2.3), rlang (>= 1.0.3), stats, tibble, tidyr (>= 1.0.0), tidyselect (>= 1.1.2), timeDate, utils, vctrs, withr

Suggests covr, ddalpha, dials (>= 1.0.0), ggplot2, igraph, kernlab, knitr, modeldata (>= 0.1.1), parsnip (>= 0.1.7), RANN, RcppRoll, rmarkdown, rpart, rsample, RSpectra, testthat (>= 3.0.0), workflows, xml2

VignetteBuilder knitr

RdMacros lifecycle

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.2.0.9000

NeedsCompilation no

Author Max Kuhn [aut, cre],
Hadley Wickham [aut],
RStudio [cph]

Maintainer Max Kuhn <`max@rstudio.com`>

Repository CRAN

Date/Publication 2022-07-07 22:30:06 UTC

R topics documented:

`add_step` *Add a New Operation to the Current Recipe*

Description

`add_step` adds a step to the last location in the recipe. `add_check` does the same for checks.

Usage

```
add_step(rec, object)

add_check(rec, object)
```

Arguments

`rec`	A `recipe()`.
`object`	A step or check object.

Value

A updated `recipe()` with the new operation in the last slot.

`bake` *Apply a trained preprocessing recipe*

Description

For a recipe with at least one preprocessing operation that has been trained by `prep()`, apply the computations to new data.

Usage

```
bake(object, ...)

## S3 method for class 'recipe'
bake(object, new_data, ..., composition = "tibble")
```

Arguments

`object`	A trained object such as a `recipe()` with at least one preprocessing operation.
`...`	One or more selector functions to choose which variables will be returned by the function. See `selections()` for more details. If no selectors are given, the default is to use `everything()`.
`new_data`	A data frame or tibble for whom the preprocessing will be applied. If `NULL` is given to `new_data`, the pre-processed *training data* will be returned (assuming that `prep(retain = TRUE)` was used).
`composition`	Either "tibble", "matrix", "data.frame", or "dgCMatrix" for the format of the processed data set. Note that all computations during the baking process are done in a non-sparse format. Also, note that this argument should be called **after** any selectors and the selectors should only resolve to numeric columns (otherwise an error is thrown).

Details

`bake()` takes a trained recipe and applies its operations to a data set to create a design matrix. If you are using a recipe as a preprocessor for modeling, we **highly recommend** that you use a `workflow()` instead of manually applying a recipe (see the example in `recipe()`).

If the data set is not too large, time can be saved by using the `retain = TRUE` option of `prep()`. This stores the processed version of the training set. With this option set, `bake(object, new_data = NULL)` will return it for free.

Also, any steps with `skip = TRUE` will not be applied to the data when `bake()` is invoked with a data set in `new_data`. `bake(object, new_data = NULL)` will always have all of the steps applied.

Value

A tibble, matrix, or sparse matrix that may have different columns than the original columns in `new_data`.

See Also

`recipe()`, `prep()`

Examples

```
data(ames, package = "modeldata")

ames <- mutate(ames, Sale_Price = log10(Sale_Price))

ames_rec <-
  recipe(Sale_Price ~ ., data = ames[-(1:6), ]) %>%
  step_other(Neighborhood, threshold = 0.05) %>%
  step_dummy(all_nominal()) %>%
  step_interact(~ starts_with("Central_Air"):Year_Built) %>%
  step_ns(Longitude, Latitude, deg_free = 2) %>%
  step_zv(all_predictors()) %>%
  prep()

# return the training set (already embedded in ames_rec)
bake(ames_rec, new_data = NULL)

# apply processing to other data:
bake(ames_rec, new_data = head(ames))

# only return selected variables:
bake(ames_rec, new_data = head(ames), all_numeric_predictors())
bake(ames_rec, new_data = head(ames), starts_with(c("Longitude", "Latitude")))
```

`case-weight-helpers` *Helpers for steps with case weights*

Description

These functions can be used to do basic calculations with or without case weights.

Usage

```
get_case_weights(info, .data)

averages(x, wts = NULL, na_rm = TRUE)

medians(x, wts = NULL)
```

```
variances(x, wts = NULL, na_rm = TRUE)

correlations(x, wts = NULL, use = "everything", method = "pearson")

covariances(x, wts = NULL, use = "everything", method = "pearson")

pca_wts(x, wts = NULL)

are_weights_used(wts, unsupervised = FALSE)
```

Arguments

`info`	A data frame from the `info` argument within steps
`.data`	The training data
`x`	A numeric vector or a data frame
`wts`	A vector of case weights
`na_rm`	A logical value indicating whether `NA` values should be removed during computations.
`use`	Used by `correlations()` or `covariances()` to pass argument to `cor()` or `cov()`
`method`	Used by `correlations()` or `covariances()` to pass argument to `cor()` or `cov()`
`unsupervised`	Can the step handle unsupervised weights

Details

`get_case_weights()` is designed for developers of recipe steps, to return a column with the role of "case weight" as a vector.

For the other functions, rows with missing case weights are removed from calculations.

For `averages()` and `variances()`, missing values in the data (*not* the case weights) only affect the calculations for those rows. For `correlations()`, the correlation matrix computation first removes rows with any missing values (equal to the "complete.obs" strategy in `stats::cor()`).

`are_weights_used()` is designed for developers of recipe steps and is used inside print method to determine how printing should be done.

`case_weights` *Using case weights with recipes*

Description

Case weights are positive numeric values that may influence how much each data point has during the preprocessing. There are a variety of situations where case weights can be used.

Details

tidymodels packages differentiate *how* different types of case weights should be used during the entire data analysis process, including preprocessing data, model fitting, performance calculations, etc.

The tidymodels packages require users to convert their numeric vectors to a vector class that reflects how these should be used. For example, there are some situations where the weights should not affect operations such as centering and scaling or other preprocessing operations.

The types of weights allowed in tidymodels are:

- Frequency weights via `hardhat::frequency_weights()`
- Importance weights via `hardhat::importance_weights()`

More types can be added by request.

For recipes, we distinguish between supervised and unsupervised steps. Supervised steps use the outcome in the calculations, this type of steps will use frequency and importance weights. Unsupervised steps don't use the outcome and will only use frequency weights.

There are 3 main principles about how case weights are used within recipes. First, the data set that is passed to the `recipe()` function should already have a case weights column in it. This column can be created beforehand using `hardhat::frequency_weights()` or `hardhat::importance_weights()`. Second, There can only be 1 case weights column in a recipe at any given time. Third, You can not modify the case weights column with most of the steps or using the `update_role()` and `add_role()` functions.

These principles ensure that you experience minimal surprises when using case weights, as the steps automatically apply case weighted operations when supported. The printing method will additionally show which steps where weighted and which steps ignored the weights because they were of an incompatible type.

See Also

`frequency_weights()`, `importance_weights()`

`check_class` *Check Variable Class*

Description

`check_class` creates a *specification* of a recipe check that will check if a variable is of a designated class.

Usage

```
check_class(
  recipe,
  ...,
  role = NA,
```

```
  trained = FALSE,
  class_nm = NULL,
  allow_additional = FALSE,
  skip = FALSE,
  class_list = NULL,
  id = rand_id("class")
)
```

Arguments

`recipe`	A recipe object. The check will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this check. See `selections()` for more details.
`role`	Not used by this check since no new variables are created.
`trained`	A logical for whether the selectors in `...` have been resolved by `prep()`.
`class_nm`	A character vector that will be used in `inherits` to check the class. If `NULL` the classes will be learned in `prep`. Can contain more than one class.
`allow_additional`	If `TRUE` a variable is allowed to have additional classes to the one(s) that are checked.
`skip`	A logical. Should the check be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`class_list`	A named list of column classes. This is `NULL` until computed by `prep()`.
`id`	A character string that is unique to this check to identify it.

Details

This function can check the classes of the variables in two ways. When the `class` argument is provided it will check if all the variables specified are of the given class. If this argument is `NULL`, the check will learn the classes of each of the specified variables in `prep`. Both ways will break `bake` if the variables are not of the requested class. If a variable has multiple classes in `prep`, all the classes are checked. Please note that in `prep` the argument `strings_as_factors` defaults to `TRUE`. If the train set contains character variables the check will be break `bake` when `strings_as_factors` is `TRUE`.

Value

An updated version of `recipe` with the new check added to the sequence of any existing operations.

Tidying

When you `tidy()` this check, a tibble with columns `terms` (the selectors or variables selected) and `value` (the type) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other checks: `check_cols()`, `check_missing()`, `check_new_values()`, `check_range()`

Examples

```
library(dplyr)
data(Sacramento, package = "modeldata")

# Learn the classes on the train set
train <- Sacramento[1:500, ]
test <- Sacramento[501:nrow(Sacramento), ]
recipe(train, sqft ~ .) %>%
  check_class(everything()) %>%
  prep(train, strings_as_factors = FALSE) %>%
  bake(test)

# Manual specification
recipe(train, sqft ~ .) %>%
  check_class(sqft, class_nm = "integer") %>%
  check_class(city, zip, type, class_nm = "factor") %>%
  check_class(latitude, longitude, class_nm = "numeric") %>%
  prep(train, strings_as_factors = FALSE) %>%
  bake(test)

# By default only the classes that are specified
#   are allowed.
x_df <- tibble(time = c(Sys.time() - 60, Sys.time()))
x_df$time %>% class()
## Not run:
recipe(x_df) %>%
  check_class(time, class_nm = "POSIXt") %>%
  prep(x_df) %>%
  bake_(x_df)

## End(Not run)

# Use allow_additional = TRUE if you are fine with it
recipe(x_df) %>%
  check_class(time, class_nm = "POSIXt", allow_additional = TRUE) %>%
  prep(x_df) %>%
  bake(x_df)
```

`check_cols` *Check if all Columns are Present*

Description

`check_cols` creates a *specification* of a recipe step that will check if all the columns of the training frame are present in the new data.

Usage

```
check_cols(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = rand_id("cols")
)
```

Arguments

`recipe`	A recipe object. The check will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this check. See `selections()` for more details.
`role`	Not used by this check since no new variables are created.
`trained`	A logical for whether the selectors in `...` have been resolved by `prep()`.
`skip`	A logical. Should the check be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this check to identify it.

Details

This check will break the `bake` function if any of the specified columns is not present in the data. If the check passes, nothing is changed to the data.

Value

An updated version of `recipe` with the new check added to the sequence of any existing operations.

Tidying

When you `tidy()` this check, a tibble with columns `terms` (the selectors or variables selected) and `value` (the type) is returned.

See Also

Other checks: `check_class()`, `check_missing()`, `check_new_values()`, `check_range()`

Examples

```
data(biomass, package = "modeldata")

biomass_rec <- recipe(HHV ~ ., data = biomass) %>%
  step_rm(sample, dataset) %>%
  check_cols(contains("gen")) %>%
  step_center(all_numeric_predictors())
## Not run:
bake(biomass_rec, biomass[, c("carbon", "HHV")])

## End(Not run)
```

`check_missing` *Check for Missing Values*

Description

`check_missing` creates a *specification* of a recipe operation that will check if variables contain missing values.

Usage

```
check_missing(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("missing")
)
```

Arguments

`recipe`	A recipe object. The check will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this check. See `selections()` for more details.
`role`	Not used by this check since no new variables are created.
`trained`	A logical for whether the selectors in `...` have been resolved by `prep()`.

`columns`	A character string of variable names that will be populated (eventually) by the terms argument.
`skip`	A logical. Should the check be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this check to identify it.

Details

This check will break the `bake` function if any of the checked columns does contain `NA` values. If the check passes, nothing is changed to the data.

Value

An updated version of `recipe` with the new check added to the sequence of any existing operations.

tidy() results

When you `tidy()` this check, a tibble with column `terms` (the selectors or variables selected) is returned.

See Also

Other checks: `check_class()`, `check_cols()`, `check_new_values()`, `check_range()`

Examples

```
data(credit_data, package = "modeldata")
is.na(credit_data) %>% colSums()

# If the test passes, `new_data` is returned unaltered
recipe(credit_data) %>%
  check_missing(Age, Expenses) %>%
  prep() %>%
  bake(credit_data)

# If your training set doesn't pass, prep() will stop with an error
## Not run:
recipe(credit_data) %>%
  check_missing(Income) %>%
  prep()

## End(Not run)

# If `new_data` contain missing values, the check will stop `bake()`

train_data <- credit_data %>% dplyr::filter(Income > 150)
test_data <- credit_data %>% dplyr::filter(Income <= 150 | is.na(Income))
```

```
rp <- recipe(train_data) %>%
  check_missing(Income) %>%
  prep()

bake(rp, train_data)
## Not run:
bake(rp, test_data)

## End(Not run)
```

`check_new_values` *Check for New Values*

Description

`check_new_values` creates a *specification* of a recipe operation that will check if variables contain new values.

Usage

```
check_new_values(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = NULL,
  ignore_NA = TRUE,
  values = NULL,
  skip = FALSE,
  id = rand_id("new_values")
)
```

Arguments

`recipe`	A recipe object. The check will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this check. See `selections()` for more details.
`role`	Not used by this check since no new variables are created.
`trained`	A logical for whether the selectors in `...` have been resolved by `prep()`.
`columns`	A character string of variable names that will be populated (eventually) by the terms argument.
`ignore_NA`	A logical that indicates if we should consider missing values as value or not. Defaults to `TRUE`.

`values`	A named list with the allowed values. This is `NULL` until computed by prep.recipe().
`skip`	A logical. Should the check be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this check to identify it.

Details

This check will break the `bake` function if any of the checked columns does contain values it did not contain when `prep` was called on the recipe. If the check passes, nothing is changed to the data.

Value

An updated version of `recipe` with the new check added to the sequence of any existing operations.

Tidying

When you `tidy()` this check, a tibble with columns `terms` (the selectors or variables selected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other checks: `check_class()`, `check_cols()`, `check_missing()`, `check_range()`

Examples

```
data(credit_data, package = "modeldata")

# If the test passes, `new_data` is returned unaltered
recipe(credit_data) %>%
  check_new_values(Home) %>%
  prep() %>%
  bake(new_data = credit_data)

# If `new_data` contains values not in `x` at the [prep()] function,
# the [bake()] function will break.
## Not run:
recipe(credit_data %>% dplyr::filter(Home != "rent")) %>%
  check_new_values(Home) %>%
  prep() %>%
  bake(new_data = credit_data)

## End(Not run)
```

```
# By default missing values are ignored, so this passes.
recipe(credit_data %>% dplyr::filter(!is.na(Home))) %>%
  check_new_values(Home) %>%
  prep() %>%
  bake(credit_data)

# Use `ignore_NA = FALSE` if you consider missing values  as a value,
# that should not occur when not observed in the train set.
## Not run:
recipe(credit_data %>% dplyr::filter(!is.na(Home))) %>%
  check_new_values(Home, ignore_NA = FALSE) %>%
  prep() %>%
  bake(credit_data)

## End(Not run)
```

`check_range` *Check Range Consistency*

Description

`check_range` creates a *specification* of a recipe check that will check if the range of a numeric variable changed in the new data.

Usage

```
check_range(
  recipe,
  ...,
  role = NA,
  skip = FALSE,
  trained = FALSE,
  slack_prop = 0.05,
  warn = FALSE,
  lower = NULL,
  upper = NULL,
  id = rand_id("range_check_")
)
```

Arguments

`recipe`	A recipe object. The check will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this check. See `selections()` for more details.
`role`	Not used by this check since no new variables are created.

`skip`	A logical. Should the check be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`trained`	A logical for whether the selectors in `...` have been resolved by `prep()`.
`slack_prop`	The allowed slack as a proportion of the range of the variable in the train set.
`warn`	If `TRUE` the check will throw a warning instead of an error when failing.
`lower`	A named numeric vector of minimum values in the train set. This is `NULL` until computed by `prep()`.
`upper`	A named numeric vector of maximum values in the train set. This is `NULL` until computed by `prep()`.
`id`	A character string that is unique to this check to identify it.

Details

The amount of slack that is allowed is determined by the `slack_prop`. This is a numeric of length one or two. If of length one, the same proportion will be used at both ends of the train set range. If of length two, its first value is used to compute the allowed slack at the lower end, the second to compute the allowed slack at the upper end.

Value

An updated version of `recipe` with the new check added to the sequence of any existing operations.

Tidying

When you `tidy()` this check, a tibble with columns `terms` (the selectors or variables selected) and `value` (the means) is returned.

See Also

Other checks: `check_class()`, `check_cols()`, `check_missing()`, `check_new_values()`

Examples

```
slack_df <- data_frame(x = 0:100)
slack_new_data <- data_frame(x = -10:110)

# this will fail the check both ends
## Not run:
recipe(slack_df) %>%
  check_range(x) %>%
  prep() %>%
  bake(slack_new_data)

## End(Not run)

# this will fail the check only at the upper end
```

```
## Not run:
recipe(slack_df) %>%
  check_range(x, slack_prop = c(0.1, 0.05)) %>%
  prep() %>%
  bake(slack_new_data)

## End(Not run)

# give a warning instead of an error
## Not run:
recipe(slack_df) %>%
  check_range(x, warn = TRUE) %>%
  prep() %>%
  bake(slack_new_data)

## End(Not run)
```

`detect_step` *Detect if a particular step or check is used in a recipe*

Description

Detect if a particular step or check is used in a recipe

Usage

```
detect_step(recipe, name)
```

Arguments

`recipe`	A recipe to check.
`name`	Character name of a step or check, omitted the prefix. That is, to check if `step_intercept` is present, use `name = intercept`.

Value

Logical indicating if recipes contains given step.

Examples

```
rec <- recipe(Species ~ ., data = iris) %>%
  step_intercept()

detect_step(rec, "intercept")
```

`discretize` *Discretize Numeric Variables*

Description

`discretize` converts a numeric vector into a factor with bins having approximately the same number of data points (based on a training set).

Usage

```
discretize(x, ...)

## Default S3 method:
discretize(x, ...)

## S3 method for class 'numeric'
discretize(
  x,
  cuts = 4,
  labels = NULL,
  prefix = "bin",
  keep_na = TRUE,
  infs = TRUE,
  min_unique = 10,
  ...
)

## S3 method for class 'discretize'
predict(object, new_data, ...)
```

Arguments

`x`	A numeric vector
`...`	Options to pass to `stats::quantile()` that should not include `x` or `probs`.
`cuts`	An integer defining how many cuts to make of the data.
`labels`	A character vector defining the factor levels that will be in the new factor (from smallest to largest). This should have length `cuts+1` and should not include a level for missing (see `keep_na` below).
`prefix`	A single parameter value to be used as a prefix for the factor levels (e.g. `bin1`, `bin2`, ...). If the string is not a valid R name, it is coerced to one. If `prefix = NULL` then the factor levels will be labelled according to the output of `cut()`.
`keep_na`	A logical for whether a factor level should be created to identify missing values in `x`. If `keep_na` is set to `TRUE` then `na.rm = TRUE` is used when calling `stats::quantile()`.
`infs`	A logical indicating whether the smallest and largest cut point should be infinite.

`min_unique`	An integer defining a sample size line of dignity for the binning. If (the number of unique values)`/(cuts+1)` is less than `min_unique`, no discretization takes place.
`object`	An object of class `discretize`.
`new_data`	A new numeric object to be binned.

Details

`discretize` estimates the cut points from `x` using percentiles. For example, if `cuts = 3`, the function estimates the quartiles of `x` and uses these as the cut points. If `cuts = 2`, the bins are defined as being above or below the median of `x`.

The `predict` method can then be used to turn numeric vectors into factor vectors.

If `keep_na = TRUE`, a suffix of "_missing" is used as a factor level (see the examples below).

If `infs = FALSE` and a new value is greater than the largest value of `x`, a missing value will result.

Value

`discretize` returns an object of class `discretize` and `predict.discretize` returns a factor vector.

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

median(biomass_tr$carbon)
discretize(biomass_tr$carbon, cuts = 2)
discretize(biomass_tr$carbon, cuts = 2, infs = FALSE)
discretize(biomass_tr$carbon, cuts = 2, infs = FALSE, keep_na = FALSE)
discretize(biomass_tr$carbon, cuts = 2, prefix = "maybe a bad idea to bin")

carbon_binned <- discretize(biomass_tr$carbon)
table(predict(carbon_binned, biomass_tr$carbon))

carbon_no_infs <- discretize(biomass_tr$carbon, infs = FALSE)
predict(carbon_no_infs, c(50, 100))
```

`formula.recipe` *Create a Formula from a Prepared Recipe*

Description

In case a model formula is required, the formula method can be used on a recipe to show what predictors and outcome(s) could be used.

Usage

```
## S3 method for class 'recipe'
formula(x, ...)
```

Arguments

`x`	A recipe object that has been prepared.
`...`	Note currently used.

Value

A formula.

Examples

```
formula(recipe(Species + Sepal.Length ~ ., data = iris) %>% prep())

iris_rec <- recipe(Species ~ ., data = iris) %>%
  step_center(all_numeric()) %>%
  prep()
formula(iris_rec)
```

`fully_trained` *Check to see if a recipe is trained/prepared*

Description

Check to see if a recipe is trained/prepared

Usage

```
fully_trained(x)
```

Arguments

`x`	A recipe

Value

A logical which is true if all of the recipe steps have been run through `prep`. If no steps have been added to the recipe, `TRUE` is returned only if the recipe has been prepped.

Examples

```
rec <- recipe(Species ~ ., data = iris) %>%
  step_center(all_numeric())

rec %>% fully_trained()


rec %>%
  prep(training = iris) %>%
  fully_trained()
```

`has_role` *Role Selection*

Description

`has_role()`, `all_predictors()`, and `all_outcomes()` can be used to select variables in a formula that have certain roles.

Similarly, `has_type()`, `all_numeric()`, and `all_nominal()` are used to select columns based on their data type. Nominal variables include both character and factor.

In most cases, the selectors `all_numeric_predictors()` and `all_nominal_predictors()`, which select on role and type, will be the right approach for users.

See selections for more details.

`current_info()` is an internal function.

All of these functions have have limited utility outside of column selection in step functions.

Usage

```
has_role(match = "predictor")

all_predictors()

all_numeric_predictors()

all_nominal_predictors()

all_outcomes()

has_type(match = "numeric")

all_numeric()

all_nominal()

current_info()
```

Arguments

`match`	A single character string for the query. Exact matching is used (i.e. regular expressions won't work).

Value

Selector functions return an integer vector.

`current_info()` returns an environment with objects `vars` and `data`.

Examples

```
data(biomass, package = "modeldata")

rec <- recipe(biomass) %>%
  update_role(
    carbon, hydrogen, oxygen, nitrogen, sulfur,
    new_role = "predictor"
  ) %>%
  update_role(HHV, new_role = "outcome") %>%
  update_role(sample, new_role = "id variable") %>%
  update_role(dataset, new_role = "splitting indicator")

recipe_info <- summary(rec)
recipe_info

# Centering on all predictors except carbon
rec %>%
  step_center(all_predictors(), -carbon) %>%
  prep(training = biomass) %>%
  bake(new_data = NULL)
```

`juice` *Extract transformed training set*

Description

As of `recipes` version 0.1.14, `juice()` **is superseded** in favor of `bake(object, new_data = NULL)`.

Usage

```
juice(object, ..., composition = "tibble")
```

Arguments

`object`	A `recipe` object that has been prepared with the option `retain = TRUE`.
`...`	One or more selector functions to choose which variables will be returned by the function. See `selections()` for more details. If no selectors are given, the default is to use `everything()`.
`composition`	Either "tibble", "matrix", "data.frame", or "dgCMatrix" for the format of the processed data set. Note that all computations during the baking process are done in a non-sparse format. Also, note that this argument should be called **after** any selectors and the selectors should only resolve to numeric columns (otherwise an error is thrown).

Details

As steps are estimated by `prep`, these operations are applied to the training set. Rather than running `bake()` to duplicate this processing, this function will return variables from the processed training set.

When preparing a recipe, if the training data set is retained using `retain = TRUE`, there is no need to `bake()` the recipe to get the preprocessed training set.

`juice()` will return the results of a recipe where *all steps* have been applied to the data, irrespective of the value of the step's `skip` argument.

See Also

`recipe()` `prep()` `bake()`

`names0` *Naming Tools*

Description

`names0` creates a series of `num` names with a common prefix. The names are numbered with leading zeros (e.g. `prefix01-prefix10` instead of `prefix1-prefix10`). `dummy_names` can be used for renaming unordered and ordered dummy variables (in `step_dummy()`).

Usage

```
names0(num, prefix = "x")

dummy_names(var, lvl, ordinal = FALSE, sep = "_")

dummy_extract_names(var, lvl, ordinal = FALSE, sep = "_")
```

Arguments

`num`	A single integer for how many elements are created.
`prefix`	A character string that will start each name.
`var`	A single string for the original factor name.
`lvl`	A character vectors of the factor levels (in order). When used with `step_dummy()`, `lvl` would be the suffixes that result *after* `model.matrix` is called (see the example below).
`ordinal`	A logical; was the original factor ordered?
`sep`	A single character value for the separator between the names and levels.

Details

When using `dummy_names()`, factor levels that are not valid variable names (e.g. "some text with spaces") will be changed to valid names by `base::make.names()`; see example below. This function will also change the names of ordinal dummy variables. Instead of values such as "`.L`", "`.Q`", or "`^4`", ordinal dummy variables are given simple integer suffixes such as "`_1`", "`_2`", etc.

Value

`names0` returns a character string of length `num` and `dummy_names` generates a character vector the same length as `lvl`.

Examples

```
names0(9, "a")
names0(10, "a")

example <- data.frame(
  x = ordered(letters[1:5]),
  y = factor(LETTERS[1:5]),
  z = factor(paste(LETTERS[1:5], 1:5))
)

dummy_names("y", levels(example$y)[-1])
dummy_names("z", levels(example$z)[-1])

after_mm <- colnames(model.matrix(~x, data = example))[-1]
after_mm
levels(example$x)

dummy_names("x", substring(after_mm, 2), ordinal = TRUE)
```

`prep` *Estimate a preprocessing recipe*

Description

For a recipe with at least one preprocessing operation, estimate the required parameters from a training set that can be later applied to other data sets.

Usage

```
prep(x, ...)

## S3 method for class 'recipe'
prep(
  x,
  training = NULL,
  fresh = FALSE,
  verbose = FALSE,
  retain = TRUE,
  log_changes = FALSE,
  strings_as_factors = TRUE,
  ...
)
```

Arguments

`x`	an object
`...`	further arguments passed to or from other methods (not currently used).
`training`	A data frame or tibble that will be used to estimate parameters for preprocessing.
`fresh`	A logical indicating whether already trained operation should be re-trained. If `TRUE`, you should pass in a data set to the argument `training`.
`verbose`	A logical that controls whether progress is reported as operations are executed.
`retain`	A logical: should the *preprocessed* training set be saved into the `template` slot of the recipe after training? This is a good idea if you want to add more steps later but want to avoid re-training the existing steps. Also, it is advisable to use `retain = TRUE` if any steps use the option `skip = FALSE`. **Note** that this can make the final recipe size large. When `verbose = TRUE`, a message is written with the approximate object size in memory but may be an underestimate since it does not take environments into account.
`log_changes`	A logical for printing a summary for each step regarding which (if any) columns were added or removed during training.
`strings_as_factors`	A logical: should character columns be converted to factors? This affects the preprocessed training set (when `retain = TRUE`) as well as the results of `bake.recipe`.

Details

Given a data set, this function estimates the required quantities and statistics needed by any operations. `prep()` returns an updated recipe with the estimates. If you are using a recipe as a preprocessor for modeling, we **highly recommend** that you use a `workflow()` instead of manually estimating a recipe (see the example in `recipe()`).

Note that missing data is handled in the steps; there is no global `na.rm` option at the recipe level or in `prep()`.

Also, if a recipe has been trained using `prep()` and then steps are added, `prep()` will only update the new operations. If `fresh = TRUE`, all of the operations will be (re)estimated.

As the steps are executed, the `training` set is updated. For example, if the first step is to center the data and the second is to scale the data, the step for scaling is given the centered data.

Value

A recipe whose step objects have been updated with the required quantities (e.g. parameter estimates, model objects, etc). Also, the `term_info` object is likely to be modified as the operations are executed.

Examples

```
data(ames, package = "modeldata")

library(dplyr)

ames <- mutate(ames, Sale_Price = log10(Sale_Price))

ames_rec <-
  recipe(
    Sale_Price ~ Longitude + Latitude + Neighborhood + Year_Built + Central_Air,
    data = ames
  ) %>%
  step_other(Neighborhood, threshold = 0.05) %>%
  step_dummy(all_nominal()) %>%
  step_interact(~ starts_with("Central_Air"):Year_Built) %>%
  step_ns(Longitude, Latitude, deg_free = 5)

prep(ames_rec, verbose = TRUE)

prep(ames_rec, log_changes = TRUE)
```

`prepper` *Wrapper function for preparing recipes within resampling*

Description

When working with the **rsample** package, a simple recipe must be *prepared* using the `prep` function first. When using recipes with **rsample** it is helpful to have a function that can prepare a recipe across a series of `split` objects that are produced in this package. `prepper` is a wrapper function around `prep` that can be used to do this. See the vignette on "Recipes and rsample" for an example.

Usage

```
prepper(split_obj, recipe, ...)
```

Arguments

`split_obj`	An `rplit` object
`recipe`	An untrained `recipe` object.
`...`	Arguments to pass to `prep` such as `verbose` or `retain`.

Details

`prepper()` sets the underlying `prep()` argument `fresh` to `TRUE`.

`print.recipe` *Print a Recipe*

Description

Print a Recipe

Usage

```
## S3 method for class 'recipe'
print(x, form_width = 30, ...)
```

Arguments

`x`	A `recipe` object
`form_width`	The number of characters used to print the variables or terms in a formula
`...`	further arguments passed to or from other methods (not currently used).

Value

The original object (invisibly)

`recipe` *Create a recipe for preprocessing data*

Description

A recipe is a description of the steps to be applied to a data set in order to prepare it for data analysis.

Usage

```
recipe(x, ...)

## Default S3 method:
recipe(x, ...)

## S3 method for class 'data.frame'
recipe(x, formula = NULL, ..., vars = NULL, roles = NULL)

## S3 method for class 'formula'
recipe(formula, data, ...)

## S3 method for class 'matrix'
recipe(x, ...)
```

Arguments

`x, data`	A data frame or tibble of the *template* data set (see below).
`...`	Further arguments passed to or from other methods (not currently used).
`formula`	A model formula. No in-line functions should be used here (e.g. `log(x)`, `x:y`, etc.) and minus signs are not allowed. These types of transformations should be enacted using `step` functions in this package. Dots are allowed as are simple multivariate outcome terms (i.e. no need for `cbind`; see Examples). A model formula may not be the best choice for high-dimensional data with many columns, because of problems with memory.
`vars`	A character string of column names corresponding to variables that will be used in any context (see below)
`roles`	A character string (the same length of `vars`) that describes a single role that the variable will take. This value could be anything but common roles are `"outcome"`, `"predictor"`, `"case_weight"`, or `"ID"`

Details

Defining recipes:

Variables in recipes can have any type of *role*, including outcome, predictor, observation ID, case weights, stratification variables, etc.

`recipe` objects can be created in several ways. If an analysis only contains outcomes and predictors, the simplest way to create one is to use a formula (e.g. `y ~ x1 + x2`) that does not contain inline functions such as `log(x3)` (see the first example below).

Alternatively, a `recipe` object can be created by first specifying which variables in a data set should be used and then sequentially defining their roles (see the last example). This alternative is an excellent choice when the number of variables is very high, as the formula method is memory-inefficient with many variables.

There are two different types of operations that can be sequentially added to a recipe.

- **Steps** can include operations like scaling a variable, creating dummy variables or interactions, and so on. More computationally complex actions such as dimension reduction or imputation can also be specified.
- **Checks** are operations that conduct specific tests of the data. When the test is satisfied, the data are returned without issue or modification. Otherwise, an error is thrown.

If you have defined a recipe and want to see which steps are included, use the `tidy()` method on the recipe object.

Note that the data passed to `recipe()` need not be the complete data that will be used to train the steps (by `prep()`). The recipe only needs to know the names and types of data that will be used. For large data sets, `head()` could be used to pass a smaller data set to save time and memory.

Using recipes:

Once a recipe is defined, it needs to be *estimated* before being applied to data. Most recipe steps have specific quantities that must be calculated or estimated. For example, `step_normalize()` needs to compute the training set's mean for the selected columns, while `step_dummy()` needs to determine the factor levels of selected columns in order to make the appropriate indicator columns.

The two most common application of recipes are modeling and stand-alone preprocessing. How the recipe is estimated depends on how it is being used.

Modeling:

The best way to use use a recipe for modeling is via the `workflows` package. This bundles a model and preprocessor (e.g. a recipe) together and gives the user a fluent way to train the model/recipe and make predictions.

```
library(dplyr)
library(workflows)
library(recipes)
library(parsnip)

data(biomass, package = "modeldata")

# split data
biomass_tr <- biomass[biomass$dataset == "Training",]
biomass_te <- biomass[biomass$dataset == "Testing",]

# With only predictors and outcomes, use a formula:
rec <- recipe(HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
              data = biomass_tr)

# Now add preprocessing steps to the recipe:
sp_signed <-
```

```
  rec %>%
  step_normalize(all_numeric_predictors()) %>%
  step_spatialsign(all_numeric_predictors())
sp_signed
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          5
##
## Operations:
##
## Centering and scaling for all_numeric_predictors()
## Spatial sign on  all_numeric_predictors()
```

We can create a `parsnip` model, and then build a workflow with the model and recipe:

```
linear_mod <- linear_reg()

linear_sp_sign_wflow <-
  workflow() %>%
  add_model(linear_mod) %>%
  add_recipe(sp_signed)

linear_sp_sign_wflow
```

```
## == Workflow ==========================================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor ------------------------------------------------------
## 2 Recipe Steps
##
## * step_normalize()
## * step_spatialsign()
##
## -- Model -------------------------------------------------------------
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

To estimate the preprocessing steps and then fit the linear model, a single call to [`fit()`][fit.model_spec()] is used:

```
linear_sp_sign_fit <- fit(linear_sp_sign_wflow, data = biomass_tr)
```

When predicting, there is no need to do anything other than call [`predict()`][predict.model_fit()]. This preprocesses the new data in the same manner as the training set, then gives the data to the linear model prediction code:

```
predict(linear_sp_sign_fit, new_data = head(biomass_te))
```

```
## # A tibble: 6 x 1
```

```
##   .pred
##   <dbl>
## 1  18.1
## 2  17.9
## 3  17.2
## 4  18.8
## 5  19.6
## 6  14.6
```

Stand-alone use of recipes:
When using a recipe to generate data for a visualization or to troubleshoot any problems with the recipe, there are functions that can be used to estimate the recipe and apply it to new data manually.

Once a recipe has been defined, the `prep()` function can be used to estimate quantities required for the operations using a data set (a.k.a. the training data). `prep()` returns a recipe.

As an example of using PCA (perhaps to produce a plot):

```
# Define the recipe
pca_rec <-
  rec %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pca(all_numeric_predictors())
```

Now to estimate the normalization statistics and the PCA loadings:

```
pca_rec <- prep(pca_rec, training = biomass_tr)
pca_rec
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          5
##
## Training data contained 456 data points and no missing data.
##
## Operations:
##
## Centering and scaling for carbon, hydrogen, oxygen, nitrogen, s... [trained]
## PCA extraction with carbon, hydrogen, oxygen, nitrogen, su... [trained]
```

Note that the estimated recipe shows the actual column names captured by the selectors.

You can `tidy.recipe()` a recipe, either when it is prepped or unprepped, to learn more about its components.

```
tidy(pca_rec)
```

```
## # A tibble: 2 x 6
##   number operation type      trained skip  id
##    <int> <chr>     <chr>     <lgl>   <lgl> <chr>
## 1      1 step      normalize TRUE    FALSE normalize_AeYA4
## 2      2 step      pca       TRUE    FALSE pca_Zn1yz
```

You can also `tidy()` recipe *steps* with a `number` or `id` argument.

To apply the prepped recipe to a data set, the [bake()] function is used in the same manner that [`predict()`][predict.model_fit()] would be for models. This applies the estimated steps to any data set.

```
bake(pca_rec, head(biomass_te))
```

```
## # A tibble: 6 x 6
##     HHV    PC1    PC2     PC3     PC4     PC5
##   <dbl>  <dbl>  <dbl>   <dbl>   <dbl>   <dbl>
## 1  18.3 0.730   0.412  0.495   0.333   0.253
## 2  17.6 0.617  -1.41  -0.118  -0.466   0.815
## 3  17.2 0.761  -1.10   0.0550 -0.397   0.747
## 4  18.9 0.0400 -0.950 -0.158   0.405  -0.143
## 5  20.5 0.792   0.732 -0.204   0.465  -0.148
## 6  18.5 0.433   0.127  0.354  -0.0168 -0.0888
```

In general, the workflow interface to recipes is recommended for most applications. [biomass$dataset == "Training",]: R:biomass$dataset%20==%20%22Training%22, [biomass$dataset == "Testing",]: R:biomass$dataset%20==%20%22Testing%22, [fit.model_spec()]: R:fit.model_spec() [predict.model_fit()]: R:predict.model_fit() `prep()`: R:prep() `prep()`: R:prep() `tidy.recipe()`: R:tidy.recipe() `tidy.recipe()`: R:tidy.recipe() [bake()]: R:bake() [predict.model_fit()]: R:predict.model_fit()

Value

An object of class `recipe` with sub-objects:

`var_info`	A tibble containing information about the original data set columns
`term_info`	A tibble that contains the current set of terms in the data set. This initially defaults to the same data contained in `var_info`.
`steps`	A list of `step` or `check` objects that define the sequence of preprocessing operations that will be applied to data. The default value is `NULL`
`template`	A tibble of the data. This is initialized to be the same as the data given in the `data` argument but can be different after the recipe is trained.

Examples

```
# formula example with single outcome:
data(biomass, package = "modeldata")

# split data
biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

# With only predictors and outcomes, use a formula
rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

# Now add preprocessing steps to the recipe
```

```
sp_signed <- rec %>%
  step_normalize(all_numeric_predictors()) %>%
  step_spatialsign(all_numeric_predictors())
sp_signed

# ----------------------------------------------------------------------------
# formula multivariate example:
# no need for `cbind(carbon, hydrogen)` for left-hand side

multi_y <- recipe(carbon + hydrogen ~ oxygen + nitrogen + sulfur,
  data = biomass_tr
)
multi_y <- multi_y %>%
  step_center(all_numeric_predictors()) %>%
  step_scale(all_numeric_predictors())

# ----------------------------------------------------------------------------
# example using `update_role` instead of formula:
# best choice for high-dimensional data

rec <- recipe(biomass_tr) %>%
  update_role(carbon, hydrogen, oxygen, nitrogen, sulfur,
    new_role = "predictor"
  ) %>%
  update_role(HHV, new_role = "outcome") %>%
  update_role(sample, new_role = "id variable") %>%
  update_role(dataset, new_role = "splitting indicator")
rec
```

`recipes` *recipes: A package for computing and preprocessing design matrices.*

Description

The `recipes` package can be used to create design matrices for modeling and to conduct preprocessing of variables. It is meant to be a more extensive framework that R's formula method. Some differences between simple formula methods and recipes are that

1. Variables can have arbitrary roles in the analysis beyond predictors and outcomes.
2. A recipe consists of one or more steps that define actions on the variables.
3. Recipes can be defined sequentially using pipes as well as being modifiable and extensible.

Basic Functions

The three main functions are `recipe()`, `prep()`, and `bake()`.

`recipe()` defines the operations on the data and the associated roles. Once the preprocessing steps are defined, any parameters are estimated using `prep()`. Once the data are ready for transformation, the `bake()` function applies the operations.

Step Functions

These functions are used to add new actions to the recipe and have the naming convention `"step_action"`. For example, `step_center()` centers the data to have a zero mean and `step_dummy()` is used to create dummy variables.

`recipes_eval_select` *Evaluate a selection with tidyselect semantics specific to recipes*

Description

`recipes_eval_select()` is a recipes specific variant of `tidyselect::eval_select()` enhanced with the ability to recognize recipes selectors, such as `all_numeric_predictors()`. See selections for more information about the unique recipes selectors.

This is a developer tool that is only useful for creating new recipes steps.

Usage

```
recipes_eval_select(
  quos,
  data,
  info,
  ...,
  allow_rename = FALSE,
  check_case_weights = TRUE
)
```

Arguments

`quos`	A list of quosures describing the selection. This is generally the `...` argument of your step function, captured with `rlang::enquos()` and stored in the step object as the `terms` element.
`data`	A data frame to use as the context to evaluate the selection in. This is generally the `training` data passed to the `prep()` method of your step.
`info`	A data frame of term information describing each column's type and role for use with the recipes selectors. This is generally the `info` data passed to the `prep()` method of your step.
`...`	These dots are for future extensions and must be empty.
`allow_rename`	Should the renaming syntax `c(foo = bar)` be allowed? This is rarely required, and is currently only used by `step_select()`. It is unlikely that your step will need renaming capabilities.
`check_case_weights`	Should selecting case weights throw an error? Defaults to `TRUE`. This is rarely changed and only needed in `juice()`, `bake.recipe()`, `update_role()`, and `add_role()`.

Value

A named character vector containing the evaluated selection. The names are always the same as the values, except when `allow_rename = TRUE`, in which case the names reflect the new names chosen by the user.

Examples

```
library(rlang)
data(scat, package = "modeldata")

rec <- recipe(Species ~ ., data = scat)

info <- summary(rec)
info

quos <- quos(all_numeric_predictors(), where(is.factor))

recipes_eval_select(quos, scat, info)
```

`roles` *Manually Alter Roles*

Description

`update_role()` alters an existing role in the recipe or assigns an initial role to variables that do not yet have a declared role.

`add_role()` adds an *additional* role to variables that already have a role in the recipe. It does not overwrite old roles, as a single variable can have multiple roles.

`remove_role()` eliminates a single existing role in the recipe.

Usage

```
add_role(recipe, ..., new_role = "predictor", new_type = NULL)

update_role(recipe, ..., new_role = "predictor", old_role = NULL)

remove_role(recipe, ..., old_role)
```

Arguments

`recipe`	An existing `recipe()`.
`...`	One or more selector functions to choose which variables are being assigned a role. See `selections()` for more details.
`new_role`	A character string for a single role.

`new_type`	A character string for specific type that the variable should be identified as. If left as `NULL`, the type is automatically identified as the *first* type you see for that variable in `summary(recipe)`.
`old_role`	A character string for the specific role to update for the variables selected by `...`. `update_role()` accepts a `NULL` as long as the variables have only a single role.

Details

Variables can have any arbitrary role (see the examples) but there are two special standard roles, `"predictor"` and `"outcome"`. These two roles are typically required when fitting a model.

`update_role()` should be used when a variable doesn't currently have a role in the recipe, or to replace an `old_role` with a `new_role`. `add_role()` only adds additional roles to variables that already have roles and will throw an error when the current role is missing (i.e. `NA`).

When using `add_role()`, if a variable is selected that already has the `new_role`, a warning is emitted and that variable is skipped so no duplicate roles are added.

Adding or updating roles is a useful way to group certain variables that don't fall in the standard `"predictor"` bucket. You can perform a step on all of the variables that have a custom role with the selector `has_role()`.

Effects of non-standard roles:

Recipes can label and retain column(s) of your data set that should not be treated as outcomes or predictors. A unique identifier column or some other ancillary data could be used to troubleshoot issues during model development but may not be either an outcome or predictor.

For example, the `modeldata::biomass` dataset has a column named `sample` with information about the specific sample type. We can change that role:

```
library(recipes)

data(biomass, package = "modeldata")
biomass_train <- biomass[1:100,]
biomass_test <- biomass[101:200,]

rec <- recipe(HHV ~ ., data = biomass_train) %>%
  update_role(sample, new_role = "id variable") %>%
  step_center(carbon)

rec <- prep(rec, biomass_train)
```

This means that `sample` is no longer treated as a `"predictor"` (the default role for columns on the right-hand side of the formula supplied to `recipe()`) and won't be used in model fitting or analysis, but will still be retained in the data set.

If you really aren't using `sample` in your recipe, we recommend that you instead remove `sample` from your dataset before passing it to `recipe()`. The reason for this is because recipes assumes that all non-standard roles are required at `bake()` time (or `predict()` time, if you are using a workflow). Since you didn't use `sample` in any steps of the recipe, you might think that you don't need to pass it to `bake()`, but this isn't true because recipes doesn't know that you didn't use it:

```
biomass_test$sample <- NULL

bake(rec, biomass_test)
```

```
#> Error in `bake()`:
#> ! The following required columns are missing from `new_data`: "sample".
#> i These columns have one of the following roles, which are required at `bake()` time: "id variable".
#> i If these roles are not required at `bake()` time, use `update_role_requirements(role = "your_role"
```

As we mentioned before, the best way to avoid this issue is to not even use a role, just remove the `sample` column from `biomass` before calling `recipe()`. In general, predictors and non-standard roles that are supplied to `recipe()` should be present at both `prep()` and `bake()` time.

If you can't remove `sample` for some reason, then the second best way to get around this issue is to tell recipes that the `"id variable"` role isn't required at `bake()` time. You can do that by using `update_role_requirements()`:

```
rec <- recipe(HHV ~ ., data = biomass_train) %>%
  update_role(sample, new_role = "id variable") %>%
  update_role_requirements("id variable", bake = FALSE) %>%
  step_center(carbon)

rec <- prep(rec, biomass_train)

# No errors!
biomass_test_baked <- bake(rec, biomass_test)
```

It should be very rare that you need this feature.

Value

An updated recipe object.

Examples

```
library(recipes)
data(biomass, package = "modeldata")

# Using the formula method, roles are created for any outcomes and predictors:
recipe(HHV ~ ., data = biomass) %>%
  summary()

# However `sample` and `dataset` aren't predictors. Since they already have
# roles, `update_role()` can be used to make changes, to any arbitrary role:
recipe(HHV ~ ., data = biomass) %>%
  update_role(sample, new_role = "id variable") %>%
  update_role(dataset, new_role = "splitting variable") %>%
  summary()

# `update_role()` cannot set a role to NA, use `remove_role()` for that
## Not run:
recipe(HHV ~ ., data = biomass) %>%
```

```
  update_role(sample, new_role = NA_character_)

## End(Not run)

# ---------------------------------------------------------------------------

# Variables can have more than one role. `add_role()` can be used
# if the column already has at least one role:
recipe(HHV ~ ., data = biomass) %>%
  add_role(carbon, sulfur, new_role = "something") %>%
  summary()

# `update_role()` has an argument called `old_role` that is required to
# unambiguously update a role when the column currently has multiple roles.
recipe(HHV ~ ., data = biomass) %>%
  add_role(carbon, new_role = "something") %>%
  update_role(carbon, new_role = "something else", old_role = "something") %>%
  summary()

# `carbon` has two roles at the end, so the last `update_roles()` fails since
# `old_role` was not given.
## Not run:
recipe(HHV ~ ., data = biomass) %>%
  add_role(carbon, sulfur, new_role = "something") %>%
  update_role(carbon, new_role = "something else")

## End(Not run)

# ---------------------------------------------------------------------------

# To remove a role, `remove_role()` can be used to remove a single role.
recipe(HHV ~ ., data = biomass) %>%
  add_role(carbon, new_role = "something") %>%
  remove_role(carbon, old_role = "something") %>%
  summary()

# To remove all roles, call `remove_role()` multiple times to reset to `NA`
recipe(HHV ~ ., data = biomass) %>%
  add_role(carbon, new_role = "something") %>%
  remove_role(carbon, old_role = "something") %>%
  remove_role(carbon, old_role = "predictor") %>%
  summary()

# ---------------------------------------------------------------------------

# If the formula method is not used, all columns have a missing role:
recipe(biomass) %>%
  summary()
```

`selections` *Methods for selecting variables in step functions*

Description

Tips for selecting columns in step functions.

Details

When selecting variables or model terms in `step` functions, `dplyr`-like tools are used. The *selector* functions can choose variables based on their name, current role, data type, or any combination of these. The selectors are passed as any other argument to the step. If the variables are explicitly named in the step function, this might look like:

```
recipe( ~ ., data = USArrests) %>%
  step_pca(Murder, Assault, UrbanPop, Rape, num_comp = 3)
```

The first four arguments indicate which variables should be used in the PCA while the last argument is a specific argument to `step_pca()` about the number of components.

Note that:

1. These arguments are not evaluated until the `prep` function for the step is executed.
2. The `dplyr`-like syntax allows for negative signs to exclude variables (e.g. `-Murder`) and the set of selectors will processed in order.
3. A leading exclusion in these arguments (e.g. `-Murder`) has the effect of adding *all* variables to the list except the excluded variable(s), ignoring role information.

Select helpers from the `tidyselect` package can also be used: `tidyselect::starts_with()`, `tidyselect::ends_with()`, `tidyselect::contains()`, `tidyselect::matches()`, `tidyselect::num_range()`, `tidyselect::everything()`, `tidyselect::one_of()`, `tidyselect::all_of()`, and `tidyselect::any_of()`

For example:

```
recipe(Species ~ ., data = iris) %>%
  step_center(starts_with("Sepal"), -contains("Width"))
```

would only select `Sepal.Length`

Columns of the design matrix that may not exist when the step is coded can also be selected. For example, when using `step_pca()`, the number of columns created by feature extraction may not be known when subsequent steps are defined. In this case, using `matches("^PC")` will select all of the columns whose names start with "PC" *once those columns are created*.

There are sets of recipes-specific functions that can be used to select variables based on their role or type: `has_role()` and `has_type()`. For convenience, there are also functions that are more specific. The functions `all_numeric()` and `all_nominal()` select based on type, with nominal variables including both character and factor; the functions `all_predictors()` and `all_outcomes()` select based on role. The functions `all_numeric_predictors()` and `all_nominal_predictors()` select intersections of role and type. Any can be used in conjunction with the previous functions described for selecting variables using their names.

A selection like this:

```
data(biomass)
recipe(HHV ~ ., data = biomass) %>%
  step_center(all_numeric(), -all_outcomes())
```

is equivalent to:

```
data(biomass)
recipe(HHV ~ ., data = biomass) %>%
  step_center(all_numeric_predictors())
```

Both result in all the numeric predictors: carbon, hydrogen, oxygen, nitrogen, and sulfur.

If a role for a variable has not been defined, it will never be selected using role-specific selectors.

Interactions:

Selectors can be used in `step_interact()` in similar ways but must be embedded in a model formula (as opposed to a sequence of selectors). For example, the interaction specification could be ~ `starts_with("Species"):Sepal.Width`. This can be useful if `Species` was converted to dummy variables previously using `step_dummy()`. The implementation of `step_interact()` is special, and is more restricted than the other step functions. Only the selector functions from recipes and tidyselect are allowed. User defined selector functions will not be recognized. Additionally, the tidyselect domain specific language is not recognized here, meaning that `&`, `|`, `!`, and `-` will not work.

Tips for saving recipes and filtering columns:

When creating variable selections:

- If you are using column filtering steps, such as `step_corr()`, try to avoid hardcoding specific variable names in downstream steps in case those columns are removed by the filter. Instead, use `dplyr::any_of()` and `dplyr::all_of()`.
  - `dplyr::any_of()` will be tolerant if a column has been removed.
  - `dplyr::all_of()` will fail unless all of the columns are present in the data.
- For both of these functions, if you are going to save the recipe as a binary object to use in another R session, try to avoid referring to a vector in your workspace.
  - Preferred: `any_of(!!var_names)`
  - Avoid: `any_of(var_names)`

Some examples:

```
some_vars <- names(mtcars)[4:6]

# No filter steps, OK for not saving the recipe
rec_1 <-
  recipe(mpg ~ ., data = mtcars) %>%
  step_log(all_of(some_vars)) %>%
  prep()

# No filter steps, saving the recipe
rec_2 <-
  recipe(mpg ~ ., data = mtcars) %>%
```

```
  step_log(!!!some_vars) %>%
  prep()

# This fails since `wt` is not in the data
recipe(mpg ~ ., data = mtcars)  %>%
  step_rm(wt) %>%
  step_log(!!!some_vars) %>%
  prep()

## Error in `chr_as_locations()`:
## ! Can't subset columns that don't exist.
## x Column `wt` doesn't exist.

# Best for filters (using any_of()) and when
# saving the recipe
rec_4 <-
  recipe(mpg ~ ., data = mtcars) %>%
  step_rm(wt) %>%
  step_log(any_of(!!some_vars)) %>%
  # equal to step_log(any_of(c("hp", "drat", "wt")))
  prep()
```

`step_arrange` *Sort rows using dplyr*

Description

`step_arrange` creates a *specification* of a recipe step that will sort rows using `dplyr::arrange()`.

Usage

```
step_arrange(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  inputs = NULL,
  skip = FALSE,
  id = rand_id("arrange")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	Comma separated list of unquoted variable names. Use ‘desc()“ to sort a variable in descending order. See `dplyr::arrange()` for more details.

`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`inputs`	Quosure of values given by `...`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

When an object in the user's global environment is referenced in the expression defining the new variable(s), it is a good idea to use quasiquotation (e.g. `!!!`) to embed the value of the object in the expression (to be portable between sessions). See the examples.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` which contains the sorting variable(s) or expression(s) is returned. The expressions are text representations and are not parsable.

Case weights

The underlying operation does not allow for case weights.

See Also

Other row operation steps: `step_filter()`, `step_impute_roll()`, `step_lag()`, `step_naomit()`, `step_sample()`, `step_shuffle()`, `step_slice()`

Other dplyr steps: `step_filter()`, `step_mutate_at()`, `step_mutate()`, `step_rename_at()`, `step_rename()`, `step_sample()`, `step_select()`, `step_slice()`

Examples

```
rec <- recipe(~., data = iris) %>%
  step_arrange(desc(Sepal.Length), 1 / Petal.Length)

prepped <- prep(rec, training = iris %>% slice(1:75))
tidy(prepped, number = 1)

library(dplyr)

dplyr_train <-
  iris %>%
  as_tibble() %>%
```

```
  slice(1:75) %>%
  dplyr::arrange(desc(Sepal.Length), 1 / Petal.Length)

rec_train <- bake(prepped, new_data = NULL)
all.equal(dplyr_train, rec_train)

dplyr_test <-
  iris %>%
  as_tibble() %>%
  slice(76:150) %>%
  dplyr::arrange(desc(Sepal.Length), 1 / Petal.Length)
rec_test <- bake(prepped, iris %>% slice(76:150))
all.equal(dplyr_test, rec_test)

# When you have variables/expressions, you can create a
# list of symbols with `rlang::syms()`` and splice them in
# the call with `!!!`. See https://tidyeval.tidyverse.org

sort_vars <- c("Sepal.Length", "Petal.Length")

qq_rec <-
  recipe(~., data = iris) %>%
  # Embed the `values` object in the call using !!!
  step_arrange(!!!syms(sort_vars)) %>%
  prep(training = iris)

tidy(qq_rec, number = 1)
```

---

step_bin2factor *Create a Factors from A Dummy Variable*

Description

`step_bin2factor` creates a *specification* of a recipe step that will create a two-level factor from a single dummy variable.

Usage

```
step_bin2factor(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  levels = c("yes", "no"),
  ref_first = TRUE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("bin2factor")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`levels`	A length 2 character string that indicates the factor levels for the 1's (in the first position) and the zeros (second)
`ref_first`	Logical. Should the first level, which replaces 1's, be the factor reference level?
`columns`	A vector with the selected variable names. This is `NULL` until computed by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This operation may be useful for situations where a binary piece of information may need to be represented as categorical instead of numeric. For example, naive Bayes models would do better to have factor predictors so that the binomial distribution is modeled instead of a Gaussian probability density of numeric binary data. Note that the numeric data is only verified to be numeric (and does not count levels).

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be affected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(covers, package = "modeldata")

rec <- recipe(~description, covers) %>%
  step_regex(description, pattern = "(rock|stony)", result = "rocks") %>%
  step_regex(description, pattern = "(rock|stony)", result = "more_rocks") %>%
  step_bin2factor(rocks)

tidy(rec, number = 3)

rec <- prep(rec, training = covers)
results <- bake(rec, new_data = covers)

table(results$rocks, results$more_rocks)

tidy(rec, number = 3)
```

`step_BoxCox` *Box-Cox Transformation for Non-Negative Data*

Description

`step_BoxCox` creates a *specification* of a recipe step that will transform data using a simple Box-Cox transformation.

Usage

```
step_BoxCox(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  lambdas = NULL,
  limits = c(-5, 5),
  num_unique = 5,
  skip = FALSE,
  id = rand_id("BoxCox")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.

`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`lambdas`	A numeric vector of transformation values. This is `NULL` until computed by `prep()`.
`limits`	A length 2 numeric vector defining the range to compute the transformation parameter lambda.
`num_unique`	An integer to specify minimum required unique values to evaluate for a transformation.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The Box-Cox transformation, which requires a strictly positive variable, can be used to rescale a variable to be more similar to a normal distribution. In this package, the partial log-likelihood function is directly optimized within a reasonable set of transformation values (which can be changed by the user).

This transformation is typically done on the outcome variable using the residuals for a statistical model (such as ordinary least squares). Here, a simple null model (intercept only) is used to apply the transformation to the *predictor* variables individually. This can have the effect of making the variable distributions more symmetric.

If the transformation parameters are estimated to be very closed to the bounds, or if the optimization fails, a value of `NA` is used and no transformation is applied.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `value` (the lambda estimate) is returned.

Case weights

The underlying operation does not allow for case weights.

References

Sakia, R. M. (1992). The Box-Cox transformation technique: A review. *The Statistician*, 169-178..

See Also

Other individual transformation steps: `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
rec <- recipe(~., data = as.data.frame(state.x77))

bc_trans <- step_BoxCox(rec, all_numeric())

bc_estimates <- prep(bc_trans, training = as.data.frame(state.x77))

bc_data <- bake(bc_estimates, as.data.frame(state.x77))

plot(density(state.x77[, "Illiteracy"]), main = "before")
plot(density(bc_data$Illiteracy), main = "after")

tidy(bc_trans, number = 1)
tidy(bc_estimates, number = 1)
```

`step_bs` *B-Spline Basis Functions*

Description

`step_bs` creates a *specification* of a recipe step that will create new columns that are basis expansions of variables using B-splines.

Usage

```
step_bs(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  deg_free = NULL,
  degree = 3,
  objects = NULL,
  options = list(),
  skip = FALSE,
  id = rand_id("bs")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`deg_free`	The degrees of freedom for the spline. As the degrees of freedom for a spline increase, more flexible and complex curves can be generated. When a single degree of freedom is used, the result is a rescaled version of the original data.
`degree`	Degree of polynomial spline (integer).
`objects`	A list of `splines::bs()` objects created once the step has been trained.
`options`	A list of options for `splines::bs()` which should not include `x`, `degree`, or `df`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_bs` can create new features from a single variable that enable fitting routines to model this variable in a nonlinear manner. The extent of the possible nonlinearity is determined by the `df`, `degree`, or `knot` arguments of `splines::bs()`. The original variables are removed from the data and new columns are added. The naming convention for the new variables is `varname_bs_1` and so on.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be affected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

with_splines <- rec %>%
  step_bs(carbon, hydrogen)
with_splines <- prep(with_splines, training = biomass_tr)

expanded <- bake(with_splines, biomass_te)
expanded
```

`step_center` *Centering numeric data*

Description

`step_center` creates a *specification* of a recipe step that will normalize numeric data to have a mean of zero.

Usage

```
step_center(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  means = NULL,
  na_rm = TRUE,
  skip = FALSE,
  id = rand_id("center")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.

`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`means`	A named numeric vector of means. This is `NULL` until computed by `prep()`.
`na_rm`	A logical value indicating whether `NA` values should be removed during computations.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Centering data means that the average of a variable is subtracted from the data. `step_center` estimates the variable means from the data used in the `training` argument of `prep.recipe`. `bake.recipe` then applies the centering to new data sets using these means.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `value` (the means) is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

Other normalization steps: `step_normalize()`, `step_range()`, `step_scale()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

center_trans <- rec %>%
```

```
  step_center(carbon, contains("gen"), -hydrogen)

center_obj <- prep(center_trans, training = biomass_tr)

transformed_te <- bake(center_obj, biomass_te)

biomass_te[1:10, names(transformed_te)]
transformed_te

tidy(center_trans, number = 1)
tidy(center_obj, number = 1)
```

`step_classdist` *Distances to Class Centroids*

Description

`step_classdist` creates a *specification* of a recipe step that will convert numeric data into Mahalanobis distance measurements to the data centroid. This is done for each value of a categorical class variable.

Usage

```
step_classdist(
  recipe,
  ...,
  class,
  role = "predictor",
  trained = FALSE,
  mean_func = mean,
  cov_func = cov,
  pool = FALSE,
  log = TRUE,
  objects = NULL,
  prefix = "classdist_",
  skip = FALSE,
  id = rand_id("classdist")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`class`	A single character string that specifies a single categorical variable to be used as the class.

`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`mean_func`	A function to compute the center of the distribution.
`cov_func`	A function that computes the covariance matrix
`pool`	A logical: should the covariance matrix be computed by pooling the data for all of the classes?
`log`	A logical: should the distances be transformed by the natural log function?
`objects`	Statistics are stored here once this step has been trained by `prep()`.
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_classdist` will create a new column for every unique value of the `class` variable. The resulting variables will not replace the original values and by default have the prefix `classdist_`. The naming format can be changed using the `prefix` argument.

Note that, by default, the default covariance function requires that each class should have at least as many rows as variables listed in the `terms` argument. If `pool = TRUE`, there must be at least as many data points are variables overall.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected), `value` (the centroid of the class), and `class` is returned.

Case weights

This step performs an supervised operation that can utilize case weights. As a result, case weights are used with frequency weights as well as importance weights. For more information,, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

Other multivariate transformation steps: `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Examples

```
# in case of missing data...
mean2 <- function(x) mean(x, na.rm = TRUE)

# define naming convention
rec <- recipe(Species ~ ., data = iris) %>%
  step_classdist(all_numeric_predictors(),
    class = "Species",
    pool = FALSE, mean_func = mean2, prefix = "centroid_"
  )

# default naming
rec <- recipe(Species ~ ., data = iris) %>%
  step_classdist(all_numeric_predictors(),
    class = "Species",
    pool = FALSE, mean_func = mean2
  )

rec_dists <- prep(rec, training = iris)

dists_to_species <- bake(rec_dists, new_data = iris, everything())
## on log scale:
dist_cols <- grep("classdist", names(dists_to_species), value = TRUE)
dists_to_species[, c("Species", dist_cols)]

tidy(rec, number = 1)
tidy(rec_dists, number = 1)
```

`step_corr` *High Correlation Filter*

Description

`step_corr` creates a *specification* of a recipe step that will potentially remove variables that have large absolute correlations with other variables.

Usage

```
step_corr(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  threshold = 0.9,
  use = "pairwise.complete.obs",
  method = "pearson",
  removals = NULL,
  skip = FALSE,
```

```
  id = rand_id("corr")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`threshold`	A value for the threshold of absolute correlation values. The step will try to remove the minimum number of columns so that all the resulting absolute correlations are less than this value.
`use`	A character string for the `use` argument to the `stats::cor()` function.
`method`	A character string for the `method` argument to the `stats::cor()` function.
`removals`	A character string that contains the names of columns that should be removed. These values are not determined until `prep()` is called.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This step can potentially remove columns from the data set. This may cause issues for subsequent steps in your recipe if the missing columns are specifically referenced by name. To avoid this, see the advice in the *Tips for saving recipes and filtering columns* section of selections.

This step attempts to remove variables to keep the largest absolute correlation between the variables less than `threshold`.

When a column has a single unique value, that column will be excluded from the correlation analysis. Also, if the data set has sporadic missing values (and an inappropriate value of `use` is chosen), some columns will also be excluded from the filter.

The arguments `use` and `method` don't take effect if case weights are used in the recipe.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be removed) is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

Author(s)

Original R code for filtering algorithm by Dong Li, modified by Max Kuhn. Contributions by Reynald Lescarbeau (for original in `caret` package). Max Kuhn for the `step` function.

See Also

Other variable filter steps: `step_filter_missing()`, `step_lincomb()`, `step_nzv()`, `step_rm()`, `step_select()`, `step_zv()`

Examples

```
data(biomass, package = "modeldata")

set.seed(3535)
biomass$duplicate <- biomass$carbon + rnorm(nrow(biomass))

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur + duplicate,
  data = biomass_tr
)

corr_filter <- rec %>%
  step_corr(all_numeric_predictors(), threshold = .5)

filter_obj <- prep(corr_filter, training = biomass_tr)

filtered_te <- bake(filter_obj, biomass_te)
round(abs(cor(biomass_tr[, c(3:7, 9)])), 2)
round(abs(cor(filtered_te)), 2)

tidy(corr_filter, number = 1)
tidy(filter_obj, number = 1)
```

`step_count` *Create Counts of Patterns using Regular Expressions*

Description

`step_count` creates a *specification* of a recipe step that will create a variable that counts instances of a regular expression pattern in text.

Usage

```
step_count(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  pattern = ".",
  normalize = FALSE,
  options = list(),
  result = make.names(pattern),
  input = NULL,
  skip = FALSE,
  id = rand_id("count")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	A single selector function to choose which variable will be searched for the regex pattern. The selector should resolve to a single variable. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`pattern`	A character string containing a regular expression (or character string for `fixed = TRUE`) to be matched in the given character vector. Coerced by `as.character` to a character string if possible.
`normalize`	A logical; should the integer counts be divided by the total number of characters in the string?.
`options`	A list of options to `gregexpr()` that should not include `x` or `pattern`.
`result`	A single character value for the name of the new variable. It should be a valid column name.
`input`	A single character value for the name of the variable being searched. This is `NULL` until computed by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `result` (the new column name) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(covers, package = "modeldata")

rec <- recipe(~description, covers) %>%
  step_count(description, pattern = "(rock|stony)", result = "rocks") %>%
  step_count(description, pattern = "famil", normalize = TRUE)

rec2 <- prep(rec, training = covers)
rec2

count_values <- bake(rec2, new_data = covers)
count_values

tidy(rec, number = 1)
tidy(rec2, number = 1)
```

`step_cut` *Cut a numeric variable into a factor*

Description

`step_cut()` creates a *specification* of a recipe step that cuts a numeric variable into a factor based on provided boundary values

Usage

```
step_cut(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  breaks,
  include_outside_range = FALSE,
  skip = FALSE,
  id = rand_id("cut")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`breaks`	A numeric vector with at least one cut point.
`include_outside_range`	Logical, indicating if values outside the range in the train set should be included in the lowest or highest bucket. Defaults to `FALSE`, values outside the original range will be set to `NA`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Unlike the `base::cut()` function there is no need to specify the min and the max values in the breaks. All values before the lowest break point will end up in the first bucket, all values after the last break points will end up in the last.

`step_cut()` will call `base::cut()` in the baking step with `include.lowest` set to `TRUE`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Case weights

The underlying operation does not allow for case weights.

See Also

Other discretization steps: `step_discretize()`

Examples

```
df <- data.frame(x = 1:10, y = 5:14)
rec <- recipe(df)

# The min and max of the variable are used as boundaries
# if they exceed the breaks
rec %>%
  step_cut(x, breaks = 5) %>%
  prep() %>%
  bake(df)

# You can use the same breaks on multiple variables
# then for each variable the boundaries are set separately
rec %>%
  step_cut(x, y, breaks = c(6, 9)) %>%
  prep() %>%
  bake(df)

# You can keep the original variables using `step_mutate` or
# `step_mutate_at`, for transforming multiple variables at once
rec %>%
  step_mutate(x_orig = x) %>%
  step_cut(x, breaks = 5) %>%
  prep() %>%
  bake(df)

# It is up to you if you want values outside the
# range learned at prep to be included
new_df <- data.frame(x = 1:11, y = 5:15)
rec %>%
  step_cut(x, breaks = 5, include_outside_range = TRUE) %>%
  prep() %>%
  bake(new_df)

rec %>%
  step_cut(x, breaks = 5, include_outside_range = FALSE) %>%
  prep() %>%
  bake(new_df)
```

`step_date` *Date Feature Generator*

Description

`step_date` creates a *specification* of a recipe step that will convert date data into one or more factor or numeric variables.

Usage

```
step_date(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  features = c("dow", "month", "year"),
  abbr = TRUE,
  label = TRUE,
  ordinal = FALSE,
  locale = Sys.getlocale("LC_TIME"),
  columns = NULL,
  keep_original_cols = TRUE,
  skip = FALSE,
  id = rand_id("date")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. The selected variables should have class `Date` or `POSIXct`. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`features`	A character string that includes at least one of the following values: `month`, `dow` (day of week), `doy` (day of year), `week`, `month`, `decimal` (decimal date, e.g. 2002.197), `quarter`, `semester`, `year`.
`abbr`	A logical. Only available for features `month` or `dow`. `FALSE` will display the day of the week as an ordered factor of character strings, such as "Sunday". `TRUE` will display an abbreviated version of the label, such as "Sun". `abbr` is disregarded if `label = FALSE`.
`label`	A logical. Only available for features `month` or `dow`. `TRUE` will display the day of the week as an ordered factor of character strings, such as "Sunday." `FALSE` will display the day of the week as a number.
`ordinal`	A logical: should factors be ordered? Only available for features `month` or `dow`.
`locale`	Locale to be used for `month` and `dow`, see locales. On Linux systems you can use `system("locale -a")` to list all the installed locales. Defaults to `Sys.getlocale("LC_TIME")`.
`columns`	A character string of variables that will be used as inputs. This field is a placeholder and will be populated once `prep()` is used.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `TRUE`.

`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Unlike some other steps, `step_date` does *not* remove the original date variables by default. Set `keep_original_cols` to `FALSE` to remove them.

See `step_time()` if you want to calculate features that are smaller than days.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected), `value` (the feature names), and `ordinal` (a logical) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
library(lubridate)

examples <- data.frame(
  Dan = ymd("2002-03-04") + days(1:10),
  Stefan = ymd("2006-01-13") + days(1:10)
)
date_rec <- recipe(~ Dan + Stefan, examples) %>%
  step_date(all_predictors())

tidy(date_rec, number = 1)

date_rec <- prep(date_rec, training = examples)

date_values <- bake(date_rec, new_data = examples)
date_values
```

```
tidy(date_rec, number = 1)
```

`step_depth` *Data Depths*

Description

`step_depth` creates a *specification* of a recipe step that will convert numeric data into measurement of *data depth*. This is done for each value of a categorical class variable.

Usage

```
step_depth(
  recipe,
  ...,
  class,
  role = "predictor",
  trained = FALSE,
  metric = "halfspace",
  options = list(),
  data = NULL,
  prefix = "depth_",
  skip = FALSE,
  id = rand_id("depth")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`class`	A single character string that specifies a single categorical variable to be used as the class.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`metric`	A character string specifying the depth metric. Possible values are "potential", "halfspace", "Mahalanobis", "simplicialVolume", "spatial", and "zoonoid".
`options`	A list of options to pass to the underlying depth functions. See `ddalpha::depth.halfspace()`, `ddalpha::depth.Mahalanobis()`, `ddalpha::depth.potential()`, `ddalpha::depth.projection()`, `ddalpha::depth.simplicial()`, `ddalpha::depth.simplicialVolume()`, `ddalpha::depth.spatial` `ddalpha::depth.zonoid()`.

`data`	The training data are stored here once after `prep()` is executed.
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Data depth metrics attempt to measure how close data a data point is to the center of its distribution. There are a number of methods for calculating depth but a simple example is the inverse of the distance of a data point to the centroid of the distribution. Generally, small values indicate that a data point not close to the centroid. `step_depth` can compute a class-specific depth for a new data point based on the proximity of the new value to the training set distribution.

This step requires the **ddalpha** package. If not installed, the step will stop with a note about installing the package.

Note that the entire training set is saved to compute future depth values. The saved data have been trained (i.e. prepared) and baked (i.e. processed) up to the point before the location that `step_depth` occupies in the recipe. Also, the data requirements for the different step methods may vary. For example, using `metric = "Mahalanobis"` requires that each class should have at least as many rows as variables listed in the `terms` argument.

The function will create a new column for every unique value of the `class` variable. The resulting variables will not replace the original values and by default have the prefix `depth_`. The naming format can be changed using the `prefix` argument.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `class` is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Examples

```
# halfspace depth is the default
rec <- recipe(Species ~ ., data = iris) %>%
  step_depth(all_numeric_predictors(), class = "Species")

# use zonoid metric instead
# also, define naming convention for new columns
rec <- recipe(Species ~ ., data = iris) %>%
  step_depth(all_numeric_predictors(),
    class = "Species",
    metric = "zonoid", prefix = "zonoid_"
  )

rec_dists <- prep(rec, training = iris)

dists_to_species <- bake(rec_dists, new_data = iris)
dists_to_species

tidy(rec, number = 1)
tidy(rec_dists, number = 1)
```

`step_discretize` *Discretize Numeric Variables*

Description

`step_discretize` creates a *specification* of a recipe step that will convert numeric data into a factor with bins having approximately the same number of data points (based on a training set).

Usage

```
step_discretize(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  num_breaks = 4,
  min_unique = 10,
  objects = NULL,
  options = list(prefix = "bin"),
  skip = FALSE,
  id = rand_id("discretize")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`num_breaks`	An integer defining how many cuts to make of the data.
`min_unique`	An integer defining a sample size line of dignity for the binning. If (the number of unique values)`/(cuts+1)` is less than `min_unique`, no discretization takes place.
`objects`	The `discretize()` objects are stored here once the recipe has be trained by `prep()`.
`options`	A list of options to `discretize()`. A default is set for the argument `x`. Note that using the options `prefix` and `labels` when more than one variable is being transformed might be problematic as all variables inherit those values.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `value` (the breaks) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other discretization steps: `step_cut()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]
```

```
rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
) %>%
  step_discretize(carbon, hydrogen)

rec <- prep(rec, biomass_tr)
binned_te <- bake(rec, biomass_te)
table(binned_te$carbon)

tidy(rec, 1)
```

`step_dummy` *Create traditional dummy variables*

Description

`step_dummy()` creates a *specification* of a recipe step that will convert nominal data (e.g. character or factors) into one or more numeric binary model terms for the levels of the original data.

Usage

```
step_dummy(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  one_hot = FALSE,
  preserve = deprecated(),
  naming = dummy_names,
  levels = NULL,
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("dummy")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details. The selected variables *must* be factors.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.

`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`one_hot`	A logical. For C levels, should C dummy variables be created rather than C-1?
`preserve`	Use `keep_original_cols` to specify whether the selected column(s) should be retained (in addition to the new dummy variables).
`naming`	A function that defines the naming convention for new dummy columns. See Details below.
`levels`	A list that contains the information needed to create dummy variables for each variable contained in `terms`. This is `NULL` until the step is trained by `prep()`.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_dummy()` will create a set of binary dummy variables from a factor variable. For example, if an unordered factor column in the data set has levels of "red", "green", "blue", the dummy variable bake will create two additional columns of 0/1 data for two of those three values (and remove the original column). For ordered factors, polynomial contrasts are used to encode the numeric values.

By default, the excluded dummy variable (i.e. the reference cell) will correspond to the first level of the unordered factor being converted.

This recipe step allows for flexible naming of the resulting variables. For an unordered factor named `x`, with levels `"a"` and `"b"`, the default naming convention would be to create a new variable called `x_b`. The naming format can be changed using the `naming` argument; the function `dummy_names()` is the default.

To change the type of contrast being used, change the global contrast option via `options`.

When the factor being converted has a missing value, all of the corresponding dummy variables are also missing. See `step_unknown()` for a solution.

When data to be processed contains novel levels (i.e., not contained in the training set), a missing value is assigned to the results. See `step_other()` for an alternative.

If no columns are selected (perhaps due to an earlier `step_zv()`), `bake()` will return the data as-is (e.g. with no dummy variables).

Note that, by default, the new dummy variable column names obey the naming rules for columns. If there are levels such as "0", `dummy_names()` will put a leading "X" in front of the level (since it uses `make.names()`). This can be changed by passing in a different function to the `naming` argument for this step.

The package vignette for dummy variables and interactions has more information.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or original variables selected) and `columns` (the list of corresponding binary columns) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

`dummy_names()`

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(Sacramento, package = "modeldata")

# Original data: city has 37 levels
length(unique(Sacramento$city))
unique(Sacramento$city) %>% sort()

rec <- recipe(~ city + sqft + price, data = Sacramento)

# Default dummy coding: 36 dummy variables
dummies <- rec %>%
  step_dummy(city) %>%
  prep(training = Sacramento)

dummy_data <- bake(dummies, new_data = NULL)

dummy_data %>%
  select(starts_with("city")) %>%
  names() # level "anything" is the reference level

# Obtain the full set of 37 dummy variables using `one_hot` option
dummies_one_hot <- rec %>%
  step_dummy(city, one_hot = TRUE) %>%
  prep(training = Sacramento)

dummy_data_one_hot <- bake(dummies_one_hot, new_data = NULL)

dummy_data_one_hot %>%
  select(starts_with("city")) %>%
  names() # no reference level
```

```
tidy(dummies, number = 1)
tidy(dummies_one_hot, number = 1)
```

`step_dummy_extract` *Extract patterns from nominal data*

Description

`step_dummy_extract()` creates a *specification* of a recipe step that will convert nominal data (e.g. character or factors) into one or more integer model terms for the extracted levels.

Usage

```
step_dummy_extract(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  sep = NULL,
  pattern = NULL,
  threshold = 0,
  other = "other",
  naming = dummy_extract_names,
  levels = NULL,
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("dummy_extract")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`sep`	Character vector containing a regular expression to use for splitting. `strsplit()` is used to perform the split. `sep` takes priority if `pattern` is also specified.
`pattern`	Character vector containing a regular expression used for extraction. `gregexpr()` and `regmatches()` are used to perform pattern extraction using `perl = TRUE`.
`threshold`	A numeric value between 0 and 1, or an integer greater or equal to one. If less than one, then factor levels with a rate of occurrence in the training set below `threshold` will be pooled to `other`. If greater or equal to one, then this value is treated as a frequency and factor levels that occur less than `threshold` times will be pooled to `other`.

`other`	A single character value for the "other" category.
`naming`	A function that defines the naming convention for new dummy columns. See Details below.
`levels`	A list that contains the information needed to create dummy variables for each variable contained in `terms`. This is `NULL` until the step is trained by `prep()`.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_dummy_extract()` will create a set of integer dummy variables from a character variable by extract individual strings by either splitting or extracting then counting those to create count variables.

Note that `threshold` works in a very specific way for this step. While it is possible for one label to be present multiple times in the same row, it will only be counted once when calculating the occurrences and frequencies.

This recipe step allows for flexible naming of the resulting variables. For an unordered factor named `x`, with levels `"a"` and `"b"`, the default naming convention would be to create a new variable called `x_b`. The naming format can be changed using the `naming` argument; the function `dummy_names()` is the default.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or original variables selected) and `columns` (the list of corresponding columns) is returned. The `columns` is is ordered according the frequency in the training data set.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

`dummy_extract_names()`

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(tate_text, package = "modeldata")

dummies <- recipe(~ artist + medium, data = tate_text) %>%
  step_dummy_extract(artist, medium, sep = ", ") %>%
  prep()

dummy_data <- bake(dummies, new_data = NULL)

dummy_data %>%
  select(starts_with("medium")) %>%
  names()

# More detailed splitting
dummies_specific <- recipe(~medium, data = tate_text) %>%
  step_dummy_extract(medium, sep = "(, )|( and )|( on )") %>%
  prep()

dummy_data_specific <- bake(dummies_specific, new_data = NULL)

dummy_data_specific %>%
  select(starts_with("medium")) %>%
  names()

tidy(dummies, number = 1)
tidy(dummies_specific, number = 1)

# pattern argument can be useful to extract harder patterns
color_examples <- tibble(
  colors = c(
    "['red', 'blue']",
    "['red', 'blue', 'white']",
    "['blue', 'blue', 'blue']"
  )
)

dummies_color <- recipe(~colors, data = color_examples) %>%
  step_dummy_extract(colors, pattern = "(?<=')[^',]+(?=')") %>%
  prep()

dommies_data_color <- dummies_color %>%
  bake(new_data = NULL)

dommies_data_color
```

`step_dummy_multi_choice` *Handle levels in multiple predictors together*

Description

`step_dummy_multi_choice()` creates a *specification* of a recipe step that will convert multiple nominal data (e.g. character or factors) into one or more numeric binary model terms for the levels of the original data.

Usage

```
step_dummy_multi_choice(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  threshold = 0,
  levels = NULL,
  input = NULL,
  other = "other",
  naming = dummy_names,
  prefix = NULL,
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("dummy_multi_choice")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details. The selected variables *must* be factors.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`threshold`	A numeric value between 0 and 1, or an integer greater or equal to one. If less than one, then factor levels with a rate of occurrence in the training set below `threshold` will be pooled to `other`. If greater or equal to one, then this value is treated as a frequency and factor levels that occur less than `threshold` times will be pooled to `other`.
`levels`	A list that contains the information needed to create dummy variables for each variable contained in `terms`. This is `NULL` until the step is trained by `prep()`.

`input`	A character vector containing the names of the columns used. This is `NULL` until the step is trained by `prep()`.
`other`	A single character value for the "other" category.
`naming`	A function that defines the naming convention for new dummy columns. See Details below.
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The overall proportion (or total counts) of the categories are computed. The "other" category is used in place of any categorical levels whose individual proportion (or frequency) in the training set is less than `threshold`.

This recipe step allows for flexible naming of the resulting variables. For an unordered factor named `x`, with levels `"a"` and `"b"`, the default naming convention would be to create a new variable called `x_b`. The naming format can be changed using the `naming` argument; the function `dummy_names()` is the default.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
library(tibble)
languages <- tribble(
  ~lang_1,    ~lang_2,   ~lang_3,
  "English",  "Italian", NA,
  "Spanish",  NA,        "French",
  "Armenian", "English", "French",
```

```
  NA,         NA,        NA
)

dummy_multi_choice_rec <- recipe(~., data = languages) %>%
  step_dummy_multi_choice(starts_with("lang")) %>%
  prep()

bake(dummy_multi_choice_rec, new_data = NULL)
tidy(dummy_multi_choice_rec, number = 1)

dummy_multi_choice_rec2 <- recipe(~., data = languages) %>%
  step_dummy_multi_choice(starts_with("lang"),
    prefix = "lang",
    threshold = 0.2
  ) %>%
  prep()

bake(dummy_multi_choice_rec2, new_data = NULL)
tidy(dummy_multi_choice_rec2, number = 1)
```

`step_factor2string` *Convert Factors to Strings*

Description

`step_factor2string` will convert one or more factor vectors to strings.

Usage

```
step_factor2string(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = FALSE,
  skip = FALSE,
  id = rand_id("factor2string")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.

`columns`	A character string of variables that will be converted. This is `NULL` until computed by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`prep` has an option `strings_as_factors` that defaults to `TRUE`. If this step is used with the default option, the string(s() produced by this step will be converted to factors after all of the steps have been prepped.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(Sacramento, package = "modeldata")

rec <- recipe(~ city + zip, data = Sacramento)

make_string <- rec %>%
  step_factor2string(city)

make_string <- prep(make_string,
  training = Sacramento,
  strings_as_factors = FALSE
)
```

```
make_string

# note that `city` is a string in recipe output
bake(make_string, new_data = NULL) %>% head()

# ...but remains a factor in the original data
Sacramento %>% head()
```

`step_filter` *Filter rows using dplyr*

Description

`step_filter` creates a *specification* of a recipe step that will remove rows using `dplyr::filter()`.

Usage

```
step_filter(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  inputs = NULL,
  skip = TRUE,
  id = rand_id("filter")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	Logical predicates defined in terms of the variables in the data. Multiple conditions are combined with `&`. Only rows where the condition evaluates to `TRUE` are kept. See `dplyr::filter()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`inputs`	Quosure of values given by `...`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = FALSE`.
`id`	A character string that is unique to this step to identify it.

Details

When an object in the user's global environment is referenced in the expression defining the new variable(s), it is a good idea to use quasiquotation (e.g. `!!`) to embed the value of the object in the expression (to be portable between sessions). See the examples.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Row Filtering

This step can entirely remove observations (rows of data), which can have unintended and/or problematic consequences when applying the step to new data later via `bake()`. Consider whether `skip = TRUE` or `skip = FALSE` is more appropriate in any given use case. In most instances that affect the rows of the data being predicted, this step probably should not be applied at all; instead, execute operations like this outside and before starting a preprocessing `recipe()`.

Tidying

When you `tidy()` this step, a tibble with column `terms` which contains the conditional statements is returned. These expressions are text representations and are not parsable.

Case weights

The underlying operation does not allow for case weights.

See Also

Other row operation steps: `step_arrange()`, `step_impute_roll()`, `step_lag()`, `step_naomit()`, `step_sample()`, `step_shuffle()`, `step_slice()`

Other dplyr steps: `step_arrange()`, `step_mutate_at()`, `step_mutate()`, `step_rename_at()`, `step_rename()`, `step_sample()`, `step_select()`, `step_slice()`

Examples

```
rec <- recipe(~., data = iris) %>%
  step_filter(Sepal.Length > 4.5, Species == "setosa")

prepped <- prep(rec, training = iris %>% slice(1:75))

library(dplyr)

dplyr_train <-
  iris %>%
  as_tibble() %>%
  slice(1:75) %>%
  dplyr::filter(Sepal.Length > 4.5, Species == "setosa")

rec_train <- bake(prepped, new_data = NULL)
all.equal(dplyr_train, rec_train)
```

```
dplyr_test <-
  iris %>%
  as_tibble() %>%
  slice(76:150) %>%
  dplyr::filter(Sepal.Length > 4.5, Species != "setosa")
rec_test <- bake(prepped, iris %>% slice(76:150))
all.equal(dplyr_test, rec_test)

values <- c("versicolor", "virginica")

qq_rec <-
  recipe(~., data = iris) %>%
  # Embed the `values` object in the call using !!
  step_filter(Sepal.Length > 4.5, Species %in% !!values)

tidy(qq_rec, number = 1)
```

`step_filter_missing` *Missing Value Column Filter*

Description

`step_filter_missing` creates a *specification* of a recipe step that will potentially remove variables that have too many missing values.

Usage

```
step_filter_missing(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  threshold = 0.1,
  removals = NULL,
  skip = FALSE,
  id = rand_id("filter_missing")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.

`threshold`	A value for the threshold of missing values in column. The step will remove the columns where the proportion of missing values exceeds the threshold.
`removals`	A character string that contains the names of columns that should be removed. These values are not determined until `prep()` is called.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This step can potentially remove columns from the data set. This may cause issues for subsequent steps in your recipe if the missing columns are specifically referenced by name. To avoid this, see the advice in the *Tips for saving recipes and filtering columns* section of selections.

This step will remove variables if the proportion of missing values exceeds the `threshold`.

All variables with missing values will be removed for `threshold = 0`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be removed) is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

Other variable filter steps: `step_corr()`, `step_lincomb()`, `step_nzv()`, `step_rm()`, `step_select()`, `step_zv()`

Examples

```
data(credit_data, package = "modeldata")

rec <- recipe(Status ~ ., data = credit_data) %>%
  step_filter_missing(all_predictors(), threshold = 0)

filter_obj <- prep(rec)
```

```
filtered_te <- bake(filter_obj, new_data = NULL)

tidy(rec, number = 1)
tidy(filter_obj, number = 1)
```

`step_geodist` *Distance between two locations*

Description

`step_geodist` creates a *specification* of a recipe step that will calculate the distance between points on a map to a reference location.

Usage

```
step_geodist(
  recipe,
  lat = NULL,
  lon = NULL,
  role = "predictor",
  trained = FALSE,
  ref_lat = NULL,
  ref_lon = NULL,
  is_lat_lon = TRUE,
  log = FALSE,
  name = "geo_dist",
  columns = NULL,
  skip = FALSE,
  id = rand_id("geodist")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`lon, lat`	Selector functions to choose which variables are used by the step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`ref_lon, ref_lat`	Single numeric values for the location of the reference point.

`is_lat_lon`	A logical: Are coordinates in latitude and longitude? If `TRUE` the Haversine formula is used and the returned result is meters. If `FALSE` the Pythagorean formula is used. Default is `TRUE` and for recipes created from previous versions of recipes, a value of `FALSE` is used.
`log`	A logical: should the distance be transformed by the natural log function?
`name`	A single character value to use for the new predictor column. If a column exists with this name, an error is issued.
`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_geodist` uses the Pythagorean theorem to calculate Euclidean distances if `is_lat_lon` is FALSE. If `is_lat_lon` is TRUE, the Haversine formula is used to calculate the great-circle distance in meters.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns echoing the values of `lat`, `lon`, `ref_lat`, `ref_lon`, `is_lat_lon`, `name`, and `id` is returned.

Case weights

The underlying operation does not allow for case weights.

References

https://en.wikipedia.org/wiki/Haversine_formula

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Examples

```
data(Smithsonian, package = "modeldata")

# How close are the museums to Union Station?
near_station <- recipe(~., data = Smithsonian) %>%
  update_role(name, new_role = "location") %>%
  step_geodist(
    lat = latitude, lon = longitude, log = FALSE,
    ref_lat = 38.8986312, ref_lon = -77.0062457,
    is_lat_lon = TRUE
  ) %>%
  prep(training = Smithsonian)

bake(near_station, new_data = NULL) %>%
  arrange(geo_dist)

tidy(near_station, number = 1)
```

`step_harmonic` *Add sin and cos terms for harmonic analysis*

Description

`step_harmonic` creates a *specification* of a recipe step that will add sin and cos terms for harmonic analysis.

Usage

```
step_harmonic(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  frequency = NA_real_,
  cycle_size = NA_real_,
  starting_val = NA_real_,
  keep_original_cols = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("harmonic")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.

`...`	One or more selector functions to choose variables for this step. See `selections()` for more details. This will typically be a single variable.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`frequency`	A numeric vector with at least one value. The value(s) must be greater than zero and finite.
`cycle_size`	A numeric vector with at least one value that indicates the size of a single cycle. `cycle_size` should have the same units as the input variable(s).
`starting_val`	either `NA`, numeric, Date or POSIXt value(s) that indicates the reference point for the sin and cos curves for each input variable. If the value is a `Date` or `POISXt` the value is converted to numeric using `as.numeric`. This parameter may be specified to increase control over the signal phase. If `starting_val` is not specified the default is 0.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`columns`	A character string of variable names that will be populated elsewhere.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This step seeks to describe periodic components of observational data using a combination of sin and cos waves. To do this, each wave of a specified frequency is modeled using one sin and one cos term. The two terms for each frequency can then be used to estimate the amplitude and phase shift of a periodic signal in observational data. The equation relating cos waves of known frequency but unknown phase and amplitude to a sum of sin and cos terms is below:

$$A_j cos(\sigma_j t_i - \Phi_j) = C_j cos(\sigma_j t_i) + S_j sin(\sigma_j t_i)$$

Solving the equation yields C_j and S_j. the amplitude can then be obtained with:

$$A_j = \sqrt{C_j^2 + S_j^2}$$

And the phase can be obtained with:

$$\Phi_j = \arctan{(S_j/C_j)}$$

where:

- $\sigma_j = 2\pi(frequency/cycle\_size))$

- A_j is the amplitude of the j^{th} frequency
- Φ_j is the phase of the j^{th} frequency
- C_j is the coefficient of the cos term for the j^{th} frequency
- S_j is the coefficient of the sin term for the j^{th} frequency

The periodic component is specified by `frequency` and `cycle_size` parameters. The cycle size relates the specified frequency to the input column(s) units. There are multiple ways to specify a wave of given frequency, for example, a `POSIXct` input column given a `frequency` of 24 and a `cycle_size` equal to 86400 is equivalent to a `frequency` of 1.0 with `cycle_size` equal to 3600.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Case weights

The underlying operation does not allow for case weights.

References

Doran, H. E., & Quilkey, J. J. (1972). Harmonic analysis of seasonal data: some important properties. American Journal of Agricultural Economics, 54, volume 4, part 1, 646-651.

Foreman, M. G. G., & Henry, R. F. (1989). The harmonic analysis of tidal model time series. Advances in water resources, 12(3), 109-120.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
library(ggplot2, quietly = TRUE)
library(dplyr)

data(sunspot.year)
sunspots <-
  tibble(
    year = 1700:1988,
    n_sunspot = sunspot.year,
    type = "measured"
  ) %>%
  slice(1:75)

# sunspots period is around 11 years, sample spacing is one year
dat <- recipe(n_sunspot ~ year, data = sunspots) %>%
  step_harmonic(year, frequency = 1 / 11, cycle_size = 1) %>%
  prep() %>%
```

```
  bake(new_data = NULL)

fit <- lm(n_sunspot ~ year_sin_1 + year_cos_1, data = dat)

preds <- tibble(
  year = sunspots$year,
  n_sunspot = fit$fitted.values,
  type = "predicted"
)

bind_rows(sunspots, preds) %>%
  ggplot(aes(x = year, y = n_sunspot, color = type)) +
  geom_line()


# ------------------------------------------------------------------------------
# POSIXct example

date_time <-
  as.POSIXct(
    paste0(rep(1959:1997, each = 12), "-", rep(1:12, length(1959:1997)), "-01"),
    tz = "UTC"
  )

carbon_dioxide <- tibble(
  date_time = date_time,
  co2 = as.numeric(co2),
  type = "measured"
)

# yearly co2 fluctuations
dat <-
  recipe(co2 ~ date_time,
    data = carbon_dioxide
  ) %>%
  step_mutate(date_time_num = as.numeric(date_time)) %>%
  step_ns(date_time_num, deg_free = 3) %>%
  step_harmonic(date_time, frequency = 1, cycle_size = 86400 * 365.24) %>%
  prep() %>%
  bake(new_data = NULL)

fit <- lm(co2 ~ date_time_num_ns_1 + date_time_num_ns_2 +
  date_time_num_ns_3 + date_time_sin_1 +
  date_time_cos_1, data = dat)

preds <- tibble(
  date_time = date_time,
  co2 = fit$fitted.values,
  type = "predicted"
)

bind_rows(carbon_dioxide, preds) %>%
  ggplot(aes(x = date_time, y = co2, color = type)) +
```

```
geom_line()
```

`step_holiday` *Holiday Feature Generator*

Description

`step_holiday` creates a *specification* of a recipe step that will convert date data into one or more binary indicator variables for common holidays.

Usage

```
step_holiday(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  holidays = c("LaborDay", "NewYearsDay", "ChristmasDay"),
  columns = NULL,
  keep_original_cols = TRUE,
  skip = FALSE,
  id = rand_id("holiday")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. The selected variables should have class `Date` or `POSIXct`. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`holidays`	A character string that includes at least one holiday supported by the `timeDate` package. See `timeDate::listHolidays()` for a complete list.
`columns`	A character string of variables that will be used as inputs. This field is a placeholder and will be populated once `prep()` is used.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `TRUE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.

`id`	A character string that is unique to this step to identify it.

Details

Unlike some other steps, `step_holiday` does *not* remove the original date variables by default. Set `keep_original_cols` to `FALSE` to remove them.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected) and `holiday` is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

`timeDate::listHolidays()`

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
library(lubridate)

examples <- data.frame(someday = ymd("2000-12-20") + days(0:40))
holiday_rec <- recipe(~someday, examples) %>%
  step_holiday(all_predictors())

holiday_rec <- prep(holiday_rec, training = examples)
holiday_values <- bake(holiday_rec, new_data = examples)
holiday_values
```

`step_hyperbolic` *Hyperbolic Transformations*

Description

`step_hyperbolic` creates a *specification* of a recipe step that will transform data using a hyperbolic function.

Usage

```
step_hyperbolic(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  func = c("sinh", "cosh", "tanh"),
  inverse = TRUE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("hyperbolic")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`func`	A character value for the function. Valid values are "sinh", "cosh", or "tanh".
`inverse`	A logical: should the inverse function be used?
`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected), `inverse`, and `func` is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
set.seed(313)
examples <- matrix(rnorm(40), ncol = 2)
examples <- as.data.frame(examples)

rec <- recipe(~ V1 + V2, data = examples)

cos_trans <- rec %>%
  step_hyperbolic(
    all_numeric_predictors(),
    func = "cosh", inverse = FALSE
  )

cos_obj <- prep(cos_trans, training = examples)

transformed_te <- bake(cos_obj, examples)
plot(examples$V1, transformed_te$V1)

tidy(cos_trans, number = 1)
tidy(cos_obj, number = 1)
```

`step_ica` *ICA Signal Extraction*

Description

`step_ica` creates a *specification* of a recipe step that will convert numeric data into one or more independent components.

Usage

```
step_ica(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  num_comp = 5,
  options = list(method = "C"),
  seed = sample.int(10000, 5),
  res = NULL,
  columns = NULL,
  prefix = "IC",
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("ica")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`num_comp`	The number of components to retain as new predictors. If `num_comp` is greater than the number of columns or the number of possible components, a smaller value will be used. If `num_comp = 0` is set then no transformation is done and selected variables will stay unchanged.
`options`	A list of options to `fastICA::fastICA()`. No defaults are set here. **Note** that the arguments `X` and `n.comp` should not be passed here.
`seed`	A single integer to set the random number stream prior to running ICA.
`res`	The `fastICA::fastICA()` object is stored here once this preprocessing step has be trained by `prep()`.
`columns`	A character string of variable names that will be populated elsewhere.
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Independent component analysis (ICA) is a transformation of a group of variables that produces a new set of artificial features or components. ICA assumes that the variables are mixtures of a set of distinct, non-Gaussian signals and attempts to transform the data to isolate these signals. Like PCA, the components are statistically independent from one another. This means that they can be used to combat large inter-variables correlations in a data set. Also like PCA, it is advisable to center and scale the variables prior to running ICA.

This package produces components using the "FastICA" methodology (see reference below). This step requires the **dimRed** and **fastICA** packages. If not installed, the step will stop with a note about installing these packages.

The argument `num_comp` controls the number of components that will be retained (the original variables that are used to derive the components are removed from the data). The new components will have names that begin with `prefix` and a sequence of numbers. The variable names are padded with zeros. For example, if `num_comp < 10`, their names will be `IC1` - `IC9`. If `num_comp = 101`, the names would be `IC001` - `IC101`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected), `value` (the loading), and `component` is returned.

Case weights

The underlying operation does not allow for case weights.

References

Hyvarinen, A., and Oja, E. (2000). Independent component analysis: algorithms and applications. *Neural Networks*, 13(4-5), 411-430.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Examples

```
# from fastICA::fastICA
set.seed(131)
S <- matrix(runif(400), 200, 2)
A <- matrix(c(1, 1, -1, 3), 2, 2, byrow = TRUE)
X <- as.data.frame(S %*% A)

tr <- X[1:100, ]
```

```
te <- X[101:200, ]

rec <- recipe(~., data = tr)

ica_trans <- step_center(rec, V1, V2)
ica_trans <- step_scale(ica_trans, V1, V2)
ica_trans <- step_ica(ica_trans, V1, V2, num_comp = 2)

if (FALSE) {
  ica_estimates <- prep(ica_trans, training = tr)
  ica_data <- bake(ica_estimates, te)

  plot(te$V1, te$V2)
  plot(ica_data$IC1, ica_data$IC2)

  tidy(ica_trans, number = 3)
  tidy(ica_estimates, number = 3)
}
```

`step_impute_bag` *Impute via bagged trees*

Description

`step_impute_bag` creates a *specification* of a recipe step that will create bagged tree models to impute missing data.

Usage

```
step_impute_bag(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  impute_with = imp_vars(all_predictors()),
  trees = 25,
  models = NULL,
  options = list(keepX = FALSE),
  seed_val = sample.int(10^4, 1),
  skip = FALSE,
  id = rand_id("impute_bag")
)

step_bagimpute(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
```

```
  impute_with = imp_vars(all_predictors()),
  trees = 25,
  models = NULL,
  options = list(keepX = FALSE),
  seed_val = sample.int(10^4, 1),
  skip = FALSE,
  id = rand_id("impute_bag")
)

imp_vars(...)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables to be imputed. When used with `imp_vars`, these dots indicate which variables are used to predict the missing data in each variable. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`impute_with`	A call to `imp_vars` to specify which variables are used to impute the variables that can include specific variable names separated by commas or different selectors (see `selections()`). If a column is included in both lists to be imputed and to be an imputation predictor, it will be removed from the latter and not used to impute itself.
`trees`	An integer for the number of bagged trees to use in each model.
`models`	The `ipred::ipredbagg()` objects are stored here once this bagged trees have be trained by `prep()`.
`options`	A list of options to `ipred::ipredbagg()`. Defaults are set for the arguments `nbagg` and `keepX` but others can be passed in. **Note** that the arguments `X` and `y` should not be passed here.
`seed_val`	An integer used to create reproducible models. The same seed is used across all imputation models.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

For each variable requiring imputation, a bagged tree is created where the outcome is the variable of interest and the predictors are any other variables listed in the `impute_with` formula. One advantage to the bagged tree is that is can accept predictors that have missing values themselves. This imputation method can be used when the variable of interest (and predictors) are numeric or categorical. Imputed categorical variables will remain categorical. Also, integers will be imputed to integer too.

Note that if a variable that is to be imputed is also in `impute_with`, this variable will be ignored.

It is possible that missing values will still occur after imputation if a large majority (or all) of the imputing variables are also missing.

As of `recipes` 0.1.16, this function name changed from `step_bagimpute()` to `step_impute_bag()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `model` (the bagged tree object) is returned.

Case weights

The underlying operation does not allow for case weights.

References

Kuhn, M. and Johnson, K. (2013). *Applied Predictive Modeling*. Springer Verlag.

See Also

Other imputation steps: `step_impute_knn()`, `step_impute_linear()`, `step_impute_lower()`, `step_impute_mean()`, `step_impute_median()`, `step_impute_mode()`, `step_impute_roll()`

Examples

```
data("credit_data", package = "modeldata")

## missing data per column
vapply(credit_data, function(x) mean(is.na(x)), c(num = 0))

set.seed(342)
in_training <- sample(1:nrow(credit_data), 2000)

credit_tr <- credit_data[in_training, ]
credit_te <- credit_data[-in_training, ]
missing_examples <- c(14, 394, 565)

rec <- recipe(Price ~ ., data = credit_tr)
## Not run:
impute_rec <- rec %>%
  step_impute_bag(Status, Home, Marital, Job, Income, Assets, Debt)
```

```
imp_models <- prep(impute_rec, training = credit_tr)

imputed_te <- bake(imp_models, new_data = credit_te, everything())

credit_te[missing_examples, ]
imputed_te[missing_examples, names(credit_te)]

tidy(impute_rec, number = 1)
tidy(imp_models, number = 1)

## Specifying which variables to imputate with

impute_rec <- rec %>%
  step_impute_bag(Status, Home, Marital, Job, Income, Assets, Debt,
    impute_with = imp_vars(Time, Age, Expenses),
    # for quick execution, nbagg lowered
    options = list(nbagg = 5, keepX = FALSE)
  )

imp_models <- prep(impute_rec, training = credit_tr)

imputed_te <- bake(imp_models, new_data = credit_te, everything())

credit_te[missing_examples, ]
imputed_te[missing_examples, names(credit_te)]

tidy(impute_rec, number = 1)
tidy(imp_models, number = 1)

## End(Not run)
```

`step_impute_knn` *Impute via k-nearest neighbors*

Description

`step_impute_knn` creates a *specification* of a recipe step that will impute missing data using nearest neighbors.

Usage

```
step_impute_knn(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  neighbors = 5,
  impute_with = imp_vars(all_predictors()),
```

```
  options = list(nthread = 1, eps = 1e-08),
  ref_data = NULL,
  columns = NULL,
  skip = FALSE,
  id = rand_id("impute_knn")
)

step_knnimpute(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  neighbors = 5,
  impute_with = imp_vars(all_predictors()),
  options = list(nthread = 1, eps = 1e-08),
  ref_data = NULL,
  columns = NULL,
  skip = FALSE,
  id = rand_id("impute_knn")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables to be imputed. When used with `imp_vars`, these dots indicate which variables are used to predict the missing data in each variable. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`neighbors`	The number of neighbors.
`impute_with`	A call to `imp_vars` to specify which variables are used to impute the variables that can include specific variable names separated by commas or different selectors (see `selections()`). If a column is included in both lists to be imputed and to be an imputation predictor, it will be removed from the latter and not used to impute itself.
`options`	A named list of options to pass to `gower::gower_topn()`. Available options are currently `nthread` and `eps`.
`ref_data`	A tibble of data that will reflect the data preprocessing done up to the point of this imputation step. This is `NULL` until the step is trained by `prep()`.
`columns`	The column names that will be imputed and used for imputation. This is `NULL` until the step is trained by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.

`id`	A character string that is unique to this step to identify it.

Details

The step uses the training set to impute any other data sets. The only distance function available is Gower's distance which can be used for mixtures of nominal and numeric data.

Once the nearest neighbors are determined, the mode is used to predictor nominal variables and the mean is used for numeric data. Note that, if the underlying data are integer, the mean will be converted to an integer too.

Note that if a variable that is to be imputed is also in `impute_with`, this variable will be ignored.

It is possible that missing values will still occur after imputation if a large majority (or all) of the imputing variables are also missing.

As of `recipes` 0.1.16, this function name changed from `step_knnimpute()` to `step_impute_knn()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables for imputation), `predictors` (those variables used to impute), and `neighbors` is returned.

Case weights

The underlying operation does not allow for case weights.

References

Gower, C. (1971) "A general coefficient of similarity and some of its properties," Biometrics, 857-871.

See Also

Other imputation steps: `step_impute_bag()`, `step_impute_linear()`, `step_impute_lower()`, `step_impute_mean()`, `step_impute_median()`, `step_impute_mode()`, `step_impute_roll()`

Examples

```
library(recipes)
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]
biomass_te_whole <- biomass_te

# induce some missing data at random
set.seed(9039)
carb_missing <- sample(1:nrow(biomass_te), 3)
```

```
nitro_missing <- sample(1:nrow(biomass_te), 3)

biomass_te$carbon[carb_missing] <- NA
biomass_te$nitrogen[nitro_missing] <- NA

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

ratio_recipe <- rec %>%
  step_impute_knn(all_predictors(), neighbors = 3)
ratio_recipe2 <- prep(ratio_recipe, training = biomass_tr)
imputed <- bake(ratio_recipe2, biomass_te)

# how well did it work?
summary(biomass_te_whole$carbon)
cbind(
  before = biomass_te_whole$carbon[carb_missing],
  after = imputed$carbon[carb_missing]
)

summary(biomass_te_whole$nitrogen)
cbind(
  before = biomass_te_whole$nitrogen[nitro_missing],
  after = imputed$nitrogen[nitro_missing]
)

tidy(ratio_recipe, number = 1)
tidy(ratio_recipe2, number = 1)
```

`step_impute_linear` *Impute numeric variables via a linear model*

Description

`step_impute_linear` creates a *specification* of a recipe step that will create linear regression models to impute missing data.

Usage

```
step_impute_linear(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  impute_with = imp_vars(all_predictors()),
  models = NULL,
```

```
  skip = FALSE,
  id = rand_id("impute_linear")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables to be imputed; these variables **must** be of type `numeric`. When used with `imp_vars`, these dots indicate which variables are used to predict the missing data in each variable. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`impute_with`	A call to `imp_vars` to specify which variables are used to impute the variables that can include specific variable names separated by commas or different selectors (see `selections()`). If a column is included in both lists to be imputed and to be an imputation predictor, it will be removed from the latter and not used to impute itself.
`models`	The `lm()` objects are stored here once the linear models have been trained by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

For each variable requiring imputation, a linear model is fit where the outcome is the variable of interest and the predictors are any other variables listed in the `impute_with` formula. Note that if a variable that is to be imputed is also in `impute_with`, this variable will be ignored.

The variable(s) to be imputed must be of type `numeric`. The imputed values will keep the same type as their original data (i.e, model predictions are coerced to integer as needed).

Since this is a linear regression, the imputation model only uses complete cases for the training set predictors.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `model` (the bagged tree object) is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

References

Kuhn, M. and Johnson, K. (2013). *Feature Engineering and Selection* `https://bookdown.org/max/FES/handling-missing-data.html`

See Also

Other imputation steps: `step_impute_bag()`, `step_impute_knn()`, `step_impute_lower()`, `step_impute_mean()`, `step_impute_median()`, `step_impute_mode()`, `step_impute_roll()`

Examples

```
data(ames, package = "modeldata")
set.seed(393)
ames_missing <- ames
ames_missing$Longitude[sample(1:nrow(ames), 200)] <- NA

imputed_ames <-
  recipe(Sale_Price ~ ., data = ames_missing) %>%
  step_impute_linear(
    Longitude,
    impute_with = imp_vars(Latitude, Neighborhood, MS_Zoning, Alley)
  ) %>%
  prep(ames_missing)

imputed <-
  bake(imputed_ames, new_data = ames_missing) %>%
  dplyr::rename(imputed = Longitude) %>%
  bind_cols(ames %>% dplyr::select(original = Longitude)) %>%
  bind_cols(ames_missing %>% dplyr::select(Longitude)) %>%
  dplyr::filter(is.na(Longitude))

library(ggplot2)
ggplot(imputed, aes(x = original, y = imputed)) +
  geom_abline(col = "green") +
  geom_point(alpha = .3) +
  coord_equal() +
  labs(title = "Imputed Values")
```

`step_impute_lower` *Impute numeric data below the threshold of measurement*

Description

`step_impute_lower` creates a *specification* of a recipe step designed for cases where the non-negative numeric data cannot be measured below a known value. In these cases, one method for imputing the data is to substitute the truncated value by a random uniform number between zero and the truncation point.

Usage

```
step_impute_lower(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  threshold = NULL,
  skip = FALSE,
  id = rand_id("impute_lower")
)

step_lowerimpute(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  threshold = NULL,
  skip = FALSE,
  id = rand_id("impute_lower")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`threshold`	A named numeric vector of lower bounds. This is `NULL` until computed by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)).

	Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_impute_lower` estimates the variable minimums from the data used in the `training` argument of `prep.recipe`. `bake.recipe` then simulates a value for any data at the minimum with a random uniform value between zero and the minimum.

As of `recipes` 0.1.16, this function name changed from `step_lowerimpute()` to `step_impute_lower()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `value` for the estimated threshold is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other imputation steps: `step_impute_bag()`, `step_impute_knn()`, `step_impute_linear()`, `step_impute_mean()`, `step_impute_median()`, `step_impute_mode()`, `step_impute_roll()`

Examples

```
library(recipes)
data(biomass, package = "modeldata")

## Truncate some values to emulate what a lower limit of
## the measurement system might look like

biomass$carbon <- ifelse(biomass$carbon > 40, biomass$carbon, 40)
biomass$hydrogen <- ifelse(biomass$hydrogen > 5, biomass$carbon, 5)

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

impute_rec <- rec %>%
  step_impute_lower(carbon, hydrogen)
```

```
tidy(impute_rec, number = 1)

impute_rec <- prep(impute_rec, training = biomass_tr)

tidy(impute_rec, number = 1)

transformed_te <- bake(impute_rec, biomass_te)

plot(transformed_te$carbon, biomass_te$carbon,
  ylab = "pre-imputation", xlab = "imputed"
)
```

`step_impute_mean` *Impute numeric data using the mean*

Description

`step_impute_mean` creates a *specification* of a recipe step that will substitute missing values of numeric variables by the training set mean of those variables.

Usage

```
step_impute_mean(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  means = NULL,
  trim = 0,
  skip = FALSE,
  id = rand_id("impute_mean")
)

step_meanimpute(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  means = NULL,
  trim = 0,
  skip = FALSE,
  id = rand_id("impute_mean")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`means`	A named numeric vector of means. This is `NULL` until computed by `prep()`. Note that, if the original data are integers, the mean will be converted to an integer to maintain the same data type.
`trim`	The fraction (0 to 0.5) of observations to be trimmed from each end of the variables before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_impute_mean` estimates the variable means from the data used in the `training` argument of `prep.recipe`. `bake.recipe` then applies the new values to new data sets using these averages.

As of `recipes` 0.1.16, this function name changed from `step_meanimpute()` to `step_impute_mean()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `model` (the mean value) is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

Other imputation steps: `step_impute_bag()`, `step_impute_knn()`, `step_impute_linear()`, `step_impute_lower()`, `step_impute_median()`, `step_impute_mode()`, `step_impute_roll()`

Examples

```
data("credit_data", package = "modeldata")

## missing data per column
vapply(credit_data, function(x) mean(is.na(x)), c(num = 0))

set.seed(342)
in_training <- sample(1:nrow(credit_data), 2000)

credit_tr <- credit_data[in_training, ]
credit_te <- credit_data[-in_training, ]
missing_examples <- c(14, 394, 565)

rec <- recipe(Price ~ ., data = credit_tr)

impute_rec <- rec %>%
  step_impute_mean(Income, Assets, Debt)

imp_models <- prep(impute_rec, training = credit_tr)

imputed_te <- bake(imp_models, new_data = credit_te, everything())

credit_te[missing_examples, ]
imputed_te[missing_examples, names(credit_te)]

tidy(impute_rec, number = 1)
tidy(imp_models, number = 1)
```

`step_impute_median` *Impute numeric data using the median*

Description

`step_impute_median` creates a *specification* of a recipe step that will substitute missing values of numeric variables by the training set median of those variables.

Usage

```
step_impute_median(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  medians = NULL,
  skip = FALSE,
  id = rand_id("impute_median")
)
```

```
step_medianimpute(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  medians = NULL,
  skip = FALSE,
  id = rand_id("impute_median")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`medians`	A named numeric vector of medians. This is `NULL` until computed by `prep()`. Note that, if the original data are integers, the median will be converted to an integer to maintain the same data type.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_impute_median` estimates the variable medians from the data used in the `training` argument of `prep.recipe`. `bake.recipe` then applies the new values to new data sets using these medians.

As of `recipes` 0.1.16, this function name changed from `step_medianimpute()` to `step_impute_median()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `model` (themedian value) is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

Other imputation steps: `step_impute_bag()`, `step_impute_knn()`, `step_impute_linear()`, `step_impute_lower()`, `step_impute_mean()`, `step_impute_mode()`, `step_impute_roll()`

Examples

```
data("credit_data", package = "modeldata")

## missing data per column
vapply(credit_data, function(x) mean(is.na(x)), c(num = 0))

set.seed(342)
in_training <- sample(1:nrow(credit_data), 2000)

credit_tr <- credit_data[in_training, ]
credit_te <- credit_data[-in_training, ]
missing_examples <- c(14, 394, 565)

rec <- recipe(Price ~ ., data = credit_tr)

impute_rec <- rec %>%
  step_impute_median(Income, Assets, Debt)

imp_models <- prep(impute_rec, training = credit_tr)

imputed_te <- bake(imp_models, new_data = credit_te, everything())

credit_te[missing_examples, ]
imputed_te[missing_examples, names(credit_te)]

tidy(impute_rec, number = 1)
tidy(imp_models, number = 1)
```

`step_impute_mode` *Impute nominal data using the most common value*

Description

`step_impute_mode` creates a *specification* of a recipe step that will substitute missing values of nominal variables by the training set mode of those variables.

Usage

```
step_impute_mode(
  recipe,
  ...,
  role = NA,
```

```
  trained = FALSE,
  modes = NULL,
  ptype = NULL,
  skip = FALSE,
  id = rand_id("impute_mode")
)

step_modeimpute(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  modes = NULL,
  ptype = NULL,
  skip = FALSE,
  id = rand_id("impute_mode")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`modes`	A named character vector of modes. This is `NULL` until computed by `prep()`.
`ptype`	A data frame prototype to cast new data sets to. This is commonly a 0-row slice of the training set.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_impute_mode` estimates the variable modes from the data used in the `training` argument of `prep.recipe`. `bake.recipe` then applies the new values to new data sets using these values. If the training set data has more than one mode, one is selected at random.

As of `recipes` 0.1.16, this function name changed from `step_modeimpute()` to `step_impute_mode()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `model` (the mode value) is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

Other imputation steps: `step_impute_bag()`, `step_impute_knn()`, `step_impute_linear()`, `step_impute_lower()`, `step_impute_mean()`, `step_impute_median()`, `step_impute_roll()`

Examples

```
data("credit_data", package = "modeldata")

## missing data per column
vapply(credit_data, function(x) mean(is.na(x)), c(num = 0))

set.seed(342)
in_training <- sample(1:nrow(credit_data), 2000)

credit_tr <- credit_data[in_training, ]
credit_te <- credit_data[-in_training, ]
missing_examples <- c(14, 394, 565)

rec <- recipe(Price ~ ., data = credit_tr)

impute_rec <- rec %>%
  step_impute_mode(Status, Home, Marital)

imp_models <- prep(impute_rec, training = credit_tr)

imputed_te <- bake(imp_models, new_data = credit_te, everything())

table(credit_te$Home, imputed_te$Home, useNA = "always")

tidy(impute_rec, number = 1)
tidy(imp_models, number = 1)
```

`step_impute_roll` *Impute numeric data using a rolling window statistic*

Description

`step_impute_roll` creates a *specification* of a recipe step that will substitute missing values of numeric variables by the measure of location (e.g. median) within a moving window.

Usage

```
step_impute_roll(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = NULL,
  statistic = median,
  window = 5,
  skip = FALSE,
  id = rand_id("impute_roll")
)

step_rollimpute(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = NULL,
  statistic = median,
  window = 5,
  skip = FALSE,
  id = rand_id("impute_roll")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables to be imputed; these columns must be non-integer numerics (i.e., double precision). See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A named numeric vector of columns. This is `NULL` until computed by `prep()`.
`statistic`	A function with a single argument for the data to compute the imputed value. Only complete values will be passed to the function and it should return a double precision value.
`window`	The size of the window around a point to be imputed. Should be an odd integer greater than one. See Details below for a discussion of points at the ends of the series.

`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

On the tails, the window is shifted towards the ends. For example, for a 5-point window, the windows for the first four points are `1:5`, `1:5`, `1:5`, and then `2:6`.

When missing data are in the window, they are not passed to the function. If all of the data in the window are missing, a missing value is returned.

The statistics are calculated on the training set values *before* imputation. This means that if previous data within the window are missing, their imputed values are not included in the window data used for imputation. In other words, each imputation does not know anything about previous imputations in the series prior to the current point.

As of `recipes` 0.1.16, this function name changed from `step_rollimpute()` to `step_impute_roll()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `window` (the window size) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other imputation steps: `step_impute_bag()`, `step_impute_knn()`, `step_impute_linear()`, `step_impute_lower()`, `step_impute_mean()`, `step_impute_median()`, `step_impute_mode()`

Other row operation steps: `step_arrange()`, `step_filter()`, `step_lag()`, `step_naomit()`, `step_sample()`, `step_shuffle()`, `step_slice()`

Examples

```
library(lubridate)

set.seed(145)
example_data <-
  data.frame(
    day = ymd("2012-06-07") + days(1:12),
    x1 = round(runif(12), 2),
    x2 = round(runif(12), 2),
```

```
    x3 = round(runif(12), 2)
  )
example_data$x1[c(1, 5, 6)] <- NA
example_data$x2[c(1:4, 10)] <- NA

library(recipes)
seven_pt <- recipe(~., data = example_data) %>%
  update_role(day, new_role = "time_index") %>%
  step_impute_roll(all_numeric_predictors(), window = 7) %>%
  prep(training = example_data)

# The training set:
bake(seven_pt, new_data = NULL)
```

`step_indicate_na` *Create Missing Data Column Indicators*

Description

`step_indicate_na` creates a *specification* of a recipe step that will create and append additional binary columns to the dataset to indicate which observations are missing.

Usage

```
step_indicate_na(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  columns = NULL,
  prefix = "na_ind",
  skip = FALSE,
  id = rand_id("indicate_na")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character string of variable names that will be populated (eventually) by the terms argument.

`prefix`	A character string that will be the prefix to the resulting new variables. Defaults to "na_ind".
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `model` (the median value) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data("credit_data", package = "modeldata")

## missing data per column
purrr::map_dbl(credit_data, function(x) mean(is.na(x)))

set.seed(342)
in_training <- sample(1:nrow(credit_data), 2000)

credit_tr <- credit_data[in_training, ]
credit_te <- credit_data[-in_training, ]

rec <- recipe(Price ~ ., data = credit_tr)

impute_rec <- rec %>%
  step_indicate_na(Income, Assets, Debt)

imp_models <- prep(impute_rec, training = credit_tr)
```

```
imputed_te <- bake(imp_models, new_data = credit_te, everything())
```

`step_integer` *Convert values to predefined integers*

Description

`step_integer` creates a *specification* of a recipe step that will convert new data into a set of integers based on the original data values.

Usage

```
step_integer(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  strict = FALSE,
  zero_based = FALSE,
  key = NULL,
  skip = FALSE,
  id = rand_id("integer")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`strict`	A logical for whether the values should be returned as integers (as opposed to double).
`zero_based`	A logical for whether the integers should start at zero and new values be appended as the largest integer.
`key`	A list that contains the information needed to create integer variables for each variable contained in `terms`. This is `NULL` until the step is trained by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_integer` will determine the unique values of each variable from the training set (excluding missing values), order them, and then assign integers to each value. When baked, each data point is translated to its corresponding integer or a value of zero for yet unseen data (although see the `zero_based` argument above). Missing values propagate.

Factor inputs are ordered by their levels. All others are ordered by `sort`.

Despite the name, the new values are returned as numeric unless `strict = TRUE`, which will coerce the results to integers.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `value` (a *list column* with the conversion key) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(Sacramento, package = "modeldata")

sacr_tr <- Sacramento[1:100, ]
sacr_tr$sqft[1] <- NA

sacr_te <- Sacramento[101:105, ]
sacr_te$sqft[1] <- NA
sacr_te$city[1] <- "whoville"
sacr_te$city[2] <- NA

rec <- recipe(type ~ ., data = sacr_tr) %>%
  step_integer(all_predictors()) %>%
  prep(training = sacr_tr)

bake(rec, sacr_te, all_predictors())
tidy(rec, number = 1)
```

`step_interact` *Create Interaction Variables*

Description

`step_interact` creates a *specification* of a recipe step that will create new columns that are interaction terms between two or more variables.

Usage

```
step_interact(
  recipe,
  terms,
  role = "predictor",
  trained = FALSE,
  objects = NULL,
  sep = "_x_",
  skip = FALSE,
  id = rand_id("interact")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`terms`	A traditional R formula that contains interaction terms. This can include `.` and selectors. See `selections()` for more details, and consider using `tidyselect::starts_with()` when dummy variables have been created.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`objects`	A list of `terms` objects for each individual interaction.
`sep`	A character value used to delineate variables in an interaction (e.g. `var1_x_var2` instead of the more traditional `var1:var2`).
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_interact` can create interactions between variables. It is primarily intended for **numeric data**; categorical variables should probably be converted to dummy variables using `step_dummy()` prior to being used for interactions.

Unlike other step functions, the `terms` argument should be a traditional R model formula but should contain no inline functions (e.g. `log`). For example, for predictors `A`, `B`, and `C`, a formula such as `~A:B:C` can be used to make a three way interaction between the variables. If the formula contains terms other than interactions (e.g. `(A+B+C)^3`) only the interaction terms are retained for the design matrix.

The separator between the variables defaults to "`_x_`" so that the three way interaction shown previously would generate a column named `A_x_B_x_C`. This can be changed using the `sep` argument.

When dummy variables are created and are used in interactions, selectors can help specify the interactions succinctly. For example, suppose a factor column `X` gets converted to dummy variables `x_2`, `x_3`, ..., `x_6` using `step_dummy()`. If you wanted an interaction with numeric column `z`, you could create a set of specific interaction effects (e.g. `x_2:z + x_3:z` and so on) or you could use `starts_with("x_"):z`. When `prep()` evaluates this step, `starts_with("x_")` resolves to `(x_2 + x_3 + x_4 + x_5 + x_6)` so that the formula is now `(x_2 + x_3 + x_4 + x_5 + x_6):z` and all two-way interactions are created.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the interaction effects) is returned.

Case weights

The underlying operation does not allow for case weights.

Examples

```
data(penguins, package = "modeldata")
penguins <- penguins %>% na.omit()

rec <- recipe(flipper_length_mm ~ ., data = penguins)

int_mod_1 <- rec %>%
  step_interact(terms = ~ bill_depth_mm:bill_length_mm)

# specify all dummy variables succinctly with `starts_with()`
int_mod_2 <- rec %>%
  step_dummy(sex, species, island) %>%
  step_interact(terms = ~ body_mass_g:starts_with("species"))

int_mod_1 <- prep(int_mod_1, training = penguins)
int_mod_2 <- prep(int_mod_2, training = penguins)
```

```
dat_1 <- bake(int_mod_1, penguins)
dat_2 <- bake(int_mod_2, penguins)

names(dat_1)
names(dat_2)

tidy(int_mod_1, number = 1)
tidy(int_mod_2, number = 2)
```

`step_intercept` *Add intercept (or constant) column*

Description

`step_intercept` creates a *specification* of a recipe step that will add an intercept or constant term in the first column of a data matrix. `step_intercept` has defaults to *predictor* role so that it is by default called in the bake step. Be careful to avoid unintentional transformations when calling steps with `all_predictors`.

Usage

```
step_intercept(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  name = "intercept",
  value = 1,
  skip = FALSE,
  id = rand_id("intercept")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	Argument ignored; included for consistency with other step specification functions.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated. Again included only for consistency.
`name`	Character name for newly added column

`value`	A numeric constant to fill the intercept column. Defaults to 1.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Case weights

The underlying operation does not allow for case weights.

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)
rec_trans <- recipe(HHV ~ ., data = biomass_tr[, -(1:2)]) %>%
  step_intercept(value = 2) %>%
  step_scale(carbon)

rec_obj <- prep(rec_trans, training = biomass_tr)

with_intercept <- bake(rec_obj, biomass_te)
with_intercept
```

`step_inverse` *Inverse Transformation*

Description

`step_inverse` creates a *specification* of a recipe step that will inverse transform the data.

Usage

```
step_inverse(
  recipe,
  ...,
  role = NA,
  offset = 0,
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("inverse")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`offset`	An optional value to add to the data prior to logging (to avoid `1/0`).
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
set.seed(313)
examples <- matrix(runif(40), ncol = 2)
examples <- data.frame(examples)

rec <- recipe(~ X1 + X2, data = examples)

inverse_trans <- rec %>%
  step_inverse(all_numeric_predictors())

inverse_obj <- prep(inverse_trans, training = examples)

transformed_te <- bake(inverse_obj, examples)
plot(examples$X1, transformed_te$X1)

tidy(inverse_trans, number = 1)
tidy(inverse_obj, number = 1)
```

`step_invlogit` *Inverse Logit Transformation*

Description

`step_invlogit` creates a *specification* of a recipe step that will transform the data from real values to be between zero and one.

Usage

```
step_invlogit(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("invlogit")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.

`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The inverse logit transformation takes values on the real line and translates them to be between zero and one using the function `f(x) = 1/(1+exp(-x))`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

ilogit_trans <- rec %>%
  step_center(carbon, hydrogen) %>%
  step_scale(carbon, hydrogen) %>%
  step_invlogit(carbon, hydrogen)
```

```
ilogit_obj <- prep(ilogit_trans, training = biomass_tr)

transformed_te <- bake(ilogit_obj, biomass_te)
plot(biomass_te$carbon, transformed_te$carbon)
```

`step_isomap` *Isomap Embedding*

Description

`step_isomap` creates a *specification* of a recipe step that will convert numeric data into one or more new dimensions.

Usage

```
step_isomap(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  num_terms = 5,
  neighbors = 50,
  options = list(.mute = c("message", "output")),
  res = NULL,
  columns = NULL,
  prefix = "Isomap",
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("isomap")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`num_terms`	The number of isomap dimensions to retain as new predictors. If `num_terms` is greater than the number of columns or the number of possible dimensions, a smaller value will be used.
`neighbors`	The number of neighbors.

`options`	A list of options to `dimRed::Isomap()`.
`res`	The `dimRed::Isomap()` object is stored here once this preprocessing step has be trained by `prep()`.
`columns`	A character string of variable names that will be populated elsewhere.
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Isomap is a form of multidimensional scaling (MDS). MDS methods try to find a reduced set of dimensions such that the geometric distances between the original data points are preserved. This version of MDS uses nearest neighbors in the data as a method for increasing the fidelity of the new dimensions to the original data values.

This step requires the **dimRed**, **RSpectra**, **igraph**, and **RANN** packages. If not installed, the step will stop with a note about installing these packages.

It is advisable to center and scale the variables prior to running Isomap (`step_center` and `step_scale` can be used for this purpose).

The argument `num_terms` controls the number of components that will be retained (the original variables that are used to derive the components are removed from the data). The new components will have names that begin with `prefix` and a sequence of numbers. The variable names are padded with zeros. For example, if `num_terms < 10`, their names will be `Isomap1` - `Isomap9`. If `num_terms = 101`, the names would be `Isomap001` - `Isomap101`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the selectors or variables selected) is returned.

Case weights

The underlying operation does not allow for case weights.

References

De Silva, V., and Tenenbaum, J. B. (2003). Global versus local methods in nonlinear dimensionality reduction. *Advances in Neural Information Processing Systems*. 721-728.

dimRed, a framework for dimensionality reduction, https://github.com/gdkrmr

See Also

Other multivariate transformation steps: step_classdist(), step_depth(), step_geodist(), step_ica(), step_kpca_poly(), step_kpca_rbf(), step_kpca(), step_mutate_at(), step_nnmf_sparse(), step_nnmf(), step_pca(), step_pls(), step_ratio(), step_spatialsign()

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

im_trans <- rec %>%
  step_YeoJohnson(all_numeric_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_isomap(all_numeric_predictors(), neighbors = 100, num_terms = 2)

if (FALSE) {
  im_estimates <- prep(im_trans, training = biomass_tr)

  im_te <- bake(im_estimates, biomass_te)

  rng <- extendrange(c(im_te$Isomap1, im_te$Isomap2))
  plot(im_te$Isomap1, im_te$Isomap2,
    xlim = rng, ylim = rng
  )

  tidy(im_trans, number = 3)
  tidy(im_estimates, number = 3)
}
```

`step_kpca` *Kernel PCA Signal Extraction*

Description

`step_kpca` creates a *specification* of a recipe step that will convert numeric data into one or more principal components using a kernel basis expansion.

Usage

```
step_kpca(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  num_comp = 5,
  res = NULL,
  columns = NULL,
  options = list(kernel = "rbfdot", kpar = list(sigma = 0.2)),
  prefix = "kPC",
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("kpca")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`num_comp`	The number of components to retain as new predictors. If `num_comp` is greater than the number of columns or the number of possible components, a smaller value will be used. If `num_comp = 0` is set then no transformation is done and selected variables will stay unchanged.
`res`	An S4 `kernlab::kpca()` object is stored here once this preprocessing step has be trained by `prep()`.
`columns`	A character string of variable names that will be populated elsewhere.
`options`	A list of options to `kernlab::kpca()`. Defaults are set for the arguments `kernel` and `kpar` but others can be passed in. **Note** that the arguments `x` and `features` should not be passed here (or at all).
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

When performing kPCA with `step_kpca()`, you must choose the kernel function (and any important kernel parameters). This step uses the **kernlab** package; the reference below discusses the types of kernels available and their parameter(s). These specifications can be made in the `kernel` and `kpar` slots of the `options` argument to `step_kpca()`. Consider using `step_kpca_rbf()` for a radial basis function kernel or `step_kpca_poly()` for a polynomial kernel.

Kernel principal component analysis (kPCA) is an extension of a PCA analysis that conducts the calculations in a broader dimensionality defined by a kernel function. For example, if a quadratic kernel function were used, each variable would be represented by its original values as well as its square. This nonlinear mapping is used during the PCA analysis and can potentially help find better representations of the original data.

This step requires the **kernlab** package. If not installed, the step will stop with a prompt about installing the package.

As with ordinary PCA, it is important to center and scale the variables prior to computing PCA components (`step_normalize()` can be used for this purpose).

The argument `num_comp` controls the number of components that will be retained; the original variables that are used to derive the components are removed from the data when `keep_original_cols = FALSE`. The new components will have names that begin with `prefix` and a sequence of numbers. The variable names are padded with zeros. For example, if `num_comp < 10`, the new names will be `kPC1` - `kPC9`. If `num_comp = 101`, the names would be `kPC001` - `kPC101`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

tidy() results

When you `tidy()` this step, a tibble with column `terms` (the selectors or variables selected) is returned.

Case weights

The underlying operation does not allow for case weights.

References

Scholkopf, B., Smola, A., and Muller, K. (1997). Kernel principal component analysis. *Lecture Notes in Computer Science*, 1327, 583-588.

Karatzoglou, K., Smola, A., Hornik, K., and Zeileis, A. (2004). kernlab - An S4 package for kernel methods in R. *Journal of Statistical Software*, 11(1), 1-20.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

kpca_trans <- rec %>%
  step_YeoJohnson(all_numeric_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_kpca(all_numeric_predictors())

if (require(kernlab) & require(ggplot2)) {
  kpca_estimates <- prep(kpca_trans, training = biomass_tr)

  kpca_te <- bake(kpca_estimates, biomass_te)

  ggplot(kpca_te, aes(x = kPC1, y = kPC2)) +
    geom_point() +
    coord_equal()

  tidy(kpca_trans, number = 3)
  tidy(kpca_estimates, number = 3)
}
```

`step_kpca_poly` *Polynomial Kernel PCA Signal Extraction*

Description

`step_kpca_poly` creates a *specification* of a recipe step that will convert numeric data into one or more principal components using a polynomial kernel basis expansion.

Usage

```
step_kpca_poly(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  num_comp = 5,
  res = NULL,
  columns = NULL,
```

```
  degree = 2,
  scale_factor = 1,
  offset = 1,
  prefix = "kPC",
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("kpca_poly")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`num_comp`	The number of components to retain as new predictors. If `num_comp` is greater than the number of columns or the number of possible components, a smaller value will be used. If `num_comp = 0` is set then no transformation is done and selected variables will stay unchanged.
`res`	An S4 `kernlab::kpca()` object is stored here once this preprocessing step has be trained by `prep()`.
`columns`	A character string of variable names that will be populated elsewhere.
`degree, scale_factor, offset`	Numeric values for the polynomial kernel function.
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Kernel principal component analysis (kPCA) is an extension of a PCA analysis that conducts the calculations in a broader dimensionality defined by a kernel function. For example, if a quadratic kernel function were used, each variable would be represented by its original values as well as its square. This nonlinear mapping is used during the PCA analysis and can potentially help find better representations of the original data.

This step requires the **kernlab** package. If not installed, the step will stop with a prompt about installing the package.

As with ordinary PCA, it is important to center and scale the variables prior to computing PCA components (`step_normalize()` can be used for this purpose).

The argument `num_comp` controls the number of components that will be retained; the original variables that are used to derive the components are removed from the data when `keep_original_cols = FALSE`. The new components will have names that begin with `prefix` and a sequence of numbers. The variable names are padded with zeros. For example, if `num_comp < 10`, the new names will be `kPC1` - `kPC9`. If `num_comp = 101`, the names would be `kPC001` - `kPC101`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

tidy() results

When you `tidy()` this step, a tibble with column `terms` (the selectors or variables selected) is returned.

Case weights

The underlying operation does not allow for case weights.

References

Scholkopf, B., Smola, A., and Muller, K. (1997). Kernel principal component analysis. *Lecture Notes in Computer Science*, 1327, 583-588.

Karatzoglou, K., Smola, A., Hornik, K., and Zeileis, A. (2004). kernlab - An S4 package for kernel methods in R. *Journal of Statistical Software*, 11(1), 1-20.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

kpca_trans <- rec %>%
```

```
  step_YeoJohnson(all_numeric_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_kpca_poly(all_numeric_predictors())

if (require(ggplot2) & require(kernlab)) {
  kpca_estimates <- prep(kpca_trans, training = biomass_tr)

  kpca_te <- bake(kpca_estimates, biomass_te)

  ggplot(kpca_te, aes(x = kPC1, y = kPC2)) +
    geom_point() +
    coord_equal()

  tidy(kpca_trans, number = 3)
  tidy(kpca_estimates, number = 3)
}
```

`step_kpca_rbf` *Radial Basis Function Kernel PCA Signal Extraction*

Description

`step_kpca_rbf` creates a *specification* of a recipe step that will convert numeric data into one or more principal components using a radial basis function kernel basis expansion.

Usage

```
step_kpca_rbf(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  num_comp = 5,
  res = NULL,
  columns = NULL,
  sigma = 0.2,
  prefix = "kPC",
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("kpca_rbf")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.

`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`num_comp`	The number of components to retain as new predictors. If `num_comp` is greater than the number of columns or the number of possible components, a smaller value will be used. If `num_comp = 0` is set then no transformation is done and selected variables will stay unchanged.
`res`	An S4 `kernlab::kpca()` object is stored here once this preprocessing step has be trained by `prep()`.
`columns`	A character string of variable names that will be populated elsewhere.
`sigma`	A numeric value for the radial basis function parameter.
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Kernel principal component analysis (kPCA) is an extension of a PCA analysis that conducts the calculations in a broader dimensionality defined by a kernel function. For example, if a quadratic kernel function were used, each variable would be represented by its original values as well as its square. This nonlinear mapping is used during the PCA analysis and can potentially help find better representations of the original data.

This step requires the **kernlab** package. If not installed, the step will stop with a prompt about installing the package.

As with ordinary PCA, it is important to center and scale the variables prior to computing PCA components (`step_normalize()` can be used for this purpose).

The argument `num_comp` controls the number of components that will be retained; the original variables that are used to derive the components are removed from the data when `keep_original_cols = FALSE`. The new components will have names that begin with `prefix` and a sequence of numbers. The variable names are padded with zeros. For example, if `num_comp < 10`, the new names will be `kPC1` - `kPC9`. If `num_comp = 101`, the names would be `kPC001` - `kPC101`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

tidy() results

When you `tidy()` this step, a tibble with column `terms` (the selectors or variables selected) is returned.

Case weights

The underlying operation does not allow for case weights.

References

Scholkopf, B., Smola, A., and Muller, K. (1997). Kernel principal component analysis. *Lecture Notes in Computer Science*, 1327, 583-588.

Karatzoglou, K., Smola, A., Hornik, K., and Zeileis, A. (2004). kernlab - An S4 package for kernel methods in R. *Journal of Statistical Software*, 11(1), 1-20.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

kpca_trans <- rec %>%
  step_YeoJohnson(all_numeric_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_kpca_rbf(all_numeric_predictors())

if (require(ggplot2) & require(kernlab)) {
  kpca_estimates <- prep(kpca_trans, training = biomass_tr)

  kpca_te <- bake(kpca_estimates, biomass_te)

  ggplot(kpca_te, aes(x = kPC1, y = kPC2)) +
    geom_point() +
    coord_equal()

  tidy(kpca_trans, number = 3)
  tidy(kpca_estimates, number = 3)
}
```

`step_lag` *Create a lagged predictor*

Description

`step_lag` creates a *specification* of a recipe step that will add new columns of lagged data. Lagged data will by default include NA values where the lag was induced. These can be removed with `step_naomit()`, or you may specify an alternative filler value with the `default` argument.

Usage

```
step_lag(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  lag = 1,
  prefix = "lag_",
  default = NA,
  columns = NULL,
  skip = FALSE,
  id = rand_id("lag")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`lag`	A vector of positive integers. Each specified column will be lagged for each value in the vector.
`prefix`	A prefix for generated column names, default to "lag_".
`default`	Passed to `dplyr::lag`, determines what fills empty rows left by lagging (defaults to NA).
`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.

`id`	A character string that is unique to this step to identify it.

Details

The step assumes that the data are already *in the proper sequential order* for lagging.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Case weights

The underlying operation does not allow for case weights.

See Also

Other row operation steps: `step_arrange()`, `step_filter()`, `step_impute_roll()`, `step_naomit()`, `step_sample()`, `step_shuffle()`, `step_slice()`

Examples

```
n <- 10
start <- as.Date("1999/01/01")
end <- as.Date("1999/01/10")

df <- data.frame(
  x = runif(n),
  index = 1:n,
  day = seq(start, end, by = "day")
)

recipe(~., data = df) %>%
  step_lag(index, day, lag = 2:3) %>%
  prep(df) %>%
  bake(df)
```

`step_lincomb` *Linear Combination Filter*

Description

`step_lincomb` creates a *specification* of a recipe step that will potentially remove numeric variables that have linear combinations between them.

Usage

```
step_lincomb(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  max_steps = 5,
  removals = NULL,
  skip = FALSE,
  id = rand_id("lincomb")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`max_steps`	The number of times to apply the algorithm.
`removals`	A character string that contains the names of columns that should be removed. These values are not determined until `prep()` is called.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This step can potentially remove columns from the data set. This may cause issues for subsequent steps in your recipe if the missing columns are specifically referenced by name. To avoid this, see the advice in the *Tips for saving recipes and filtering columns* section of selections.

This step finds exact linear combinations between two or more variables and recommends which column(s) should be removed to resolve the issue. This algorithm may need to be applied multiple times (as defined by `max_steps`).

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be removed) is returned.

Case weights

The underlying operation does not allow for case weights.

Author(s)

Max Kuhn, Kirk Mettler, and Jed Wing

See Also

Other variable filter steps: `step_corr()`, `step_filter_missing()`, `step_nzv()`, `step_rm()`, `step_select()`, `step_zv()`

Examples

```
data(biomass, package = "modeldata")

biomass$new_1 <- with(
  biomass,
  .1 * carbon - .2 * hydrogen + .6 * sulfur
)
biomass$new_2 <- with(
  biomass,
  .5 * carbon - .2 * oxygen + .6 * nitrogen
)

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(HHV ~ carbon + hydrogen + oxygen + nitrogen +
  sulfur + new_1 + new_2,
data = biomass_tr
)

lincomb_filter <- rec %>%
  step_lincomb(all_numeric_predictors())

lincomb_filter_trained <- prep(lincomb_filter, training = biomass_tr)
lincomb_filter_trained

tidy(lincomb_filter, number = 1)
tidy(lincomb_filter_trained, number = 1)
```

`step_log` *Logarithmic Transformation*

Description

`step_log` creates a *specification* of a recipe step that will log transform data.

Usage

```
step_log(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  base = exp(1),
  offset = 0,
  columns = NULL,
  skip = FALSE,
  signed = FALSE,
  id = rand_id("log")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`base`	A numeric value for the base.
`offset`	An optional value to add to the data prior to logging (to avoid `log(0)`).
`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`signed`	A logical indicating whether to take the signed log. This is sign(x) * abs(log(x)) when abs(x) => 1 or 0 if abs(x) < 1. If `TRUE` the `offset` argument will be ignored.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected) and `base`.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
set.seed(313)
examples <- matrix(exp(rnorm(40)), ncol = 2)
examples <- as.data.frame(examples)

rec <- recipe(~ V1 + V2, data = examples)

log_trans <- rec %>%
  step_log(all_numeric_predictors())

log_obj <- prep(log_trans, training = examples)

transformed_te <- bake(log_obj, examples)
plot(examples$V1, transformed_te$V1)

tidy(log_trans, number = 1)
tidy(log_obj, number = 1)

# using the signed argument with negative values

examples2 <- matrix(rnorm(40, sd = 5), ncol = 2)
examples2 <- as.data.frame(examples2)

recipe(~ V1 + V2, data = examples2) %>%
  step_log(all_numeric_predictors()) %>%
  prep(training = examples2) %>%
  bake(examples2)

recipe(~ V1 + V2, data = examples2) %>%
  step_log(all_numeric_predictors(), signed = TRUE) %>%
  prep(training = examples2) %>%
  bake(examples2)
```

`step_logit` *Logit Transformation*

Description

`step_logit` creates a *specification* of a recipe step that will logit transform the data.

Usage

```
step_logit(
  recipe,
  ...,
  offset = 0,
  role = NA,
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("logit")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`offset`	A numeric value to modify values of the columns that are either one or zero. They are modified to be `x - offset` or `offset`, respectively.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The logit transformation takes values between zero and one and translates them to be on the real line using the function `f(p) = log(p/(1-p))`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
set.seed(313)
examples <- matrix(runif(40), ncol = 2)
examples <- data.frame(examples)

rec <- recipe(~ X1 + X2, data = examples)

logit_trans <- rec %>%
  step_logit(all_numeric_predictors())

logit_obj <- prep(logit_trans, training = examples)

transformed_te <- bake(logit_obj, examples)
plot(examples$X1, transformed_te$X1)

tidy(logit_trans, number = 1)
tidy(logit_obj, number = 1)
```

`step_mutate` *Add new variables using dplyr*

Description

`step_mutate()` creates a *specification* of a recipe step that will add variables using `dplyr::mutate()`.

Usage

```
step_mutate(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  inputs = NULL,
  skip = FALSE,
  id = rand_id("mutate")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	Name-value pairs of expressions. See `dplyr::mutate()`.

`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`inputs`	Quosure(s) of `...`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

When using this flexible step, use extra care to avoid data leakage in your preprocessing. Consider, for example, the transformation `x = w > mean(w)`. When applied to new data or testing data, this transformation would use the mean of `w` from the *new* data, not the mean of `w` from the training data.

When an object in the user's global environment is referenced in the expression defining the new variable(s), it is a good idea to use quasiquotation (e.g. `!!`) to embed the value of the object in the expression (to be portable between sessions). See the examples.

If a preceding step removes a column that is selected by name in `step_mutate()`, the recipe will error when being estimated with `prep()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `values`, which contains the `mutate()` expressions as character strings (and are not reparsable), is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Other dplyr steps: `step_arrange()`, `step_filter()`, `step_mutate_at()`, `step_rename_at()`, `step_rename()`, `step_sample()`, `step_select()`, `step_slice()`

Examples

```
rec <-
  recipe(~., data = iris) %>%
  step_mutate(
    dbl_width = Sepal.Width * 2,
    half_length = Sepal.Length / 2
  )

prepped <- prep(rec, training = iris %>% slice(1:75))

library(dplyr)

dplyr_train <-
  iris %>%
  as_tibble() %>%
  slice(1:75) %>%
  mutate(
    dbl_width = Sepal.Width * 2,
    half_length = Sepal.Length / 2
  )

rec_train <- bake(prepped, new_data = NULL)
all.equal(dplyr_train, rec_train)

dplyr_test <-
  iris %>%
  as_tibble() %>%
  slice(76:150) %>%
  mutate(
    dbl_width = Sepal.Width * 2,
    half_length = Sepal.Length / 2
  )
rec_test <- bake(prepped, iris %>% slice(76:150))
all.equal(dplyr_test, rec_test)

# Embedding objects:
const <- 1.414

qq_rec <-
  recipe(~., data = iris) %>%
  step_mutate(
    bad_approach = Sepal.Width * const,
    best_approach = Sepal.Width * !!const
  ) %>%
  prep(training = iris)

bake(qq_rec, new_data = NULL, contains("appro")) %>% slice(1:4)

# The difference:
tidy(qq_rec, number = 1)
```

`step_mutate_at` *Mutate multiple columns using dplyr*

Description

`step_mutate_at` creates a *specification* of a recipe step that will modify the selected variables using a common function via `dplyr::mutate_at()`.

Usage

```
step_mutate_at(
  recipe,
  ...,
  fn,
  role = "predictor",
  trained = FALSE,
  inputs = NULL,
  skip = FALSE,
  id = rand_id("mutate_at")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`fn`	A function fun, a quosure style lambda '~ fun(.)" or a list of either form. (see `dplyr::mutate_at()`). **Note that this argument must be named**.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`inputs`	A vector of column names populated by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

When using this flexible step, use extra care to avoid data leakage in your preprocessing. Consider, for example, the transformation `x = w > mean(w)`. When applied to new data or testing data, this transformation would use the mean of `w` from the *new* data, not the mean of `w` from the training data.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` which contains the columns being transformed is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Other dplyr steps: `step_arrange()`, `step_filter()`, `step_mutate()`, `step_rename_at()`, `step_rename()`, `step_sample()`, `step_select()`, `step_slice()`

Examples

```
library(dplyr)
recipe(~., data = iris) %>%
  step_mutate_at(contains("Length"), fn = ~ 1 / .) %>%
  prep() %>%
  bake(new_data = NULL) %>%
  slice(1:10)

recipe(~., data = iris) %>%
  # leads to more columns being created.
  step_mutate_at(contains("Length"), fn = list(log = log, sqrt = sqrt)) %>%
  prep() %>%
  bake(new_data = NULL) %>%
  slice(1:10)
```

`step_naomit` *Remove observations with missing values*

Description

`step_naomit` creates a *specification* of a recipe step that will remove observations (rows of data) if they contain `NA` or `NaN` values.

Usage

```
step_naomit(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = NULL,
  skip = TRUE,
  id = rand_id("naomit")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Unused, include for consistency with other steps.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated. Again included for consistency.
`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = FALSE`.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Row Filtering

This step can entirely remove observations (rows of data), which can have unintended and/or problematic consequences when applying the step to new data later via `bake()`. Consider whether `skip = TRUE` or `skip = FALSE` is more appropriate in any given use case. In most instances that affect the rows of the data being predicted, this step probably should not be applied at all; instead, execute operations like this outside and before starting a preprocessing `recipe()`.

Case weights

The underlying operation does not allow for case weights.

See Also

Other row operation steps: `step_arrange()`, `step_filter()`, `step_impute_roll()`, `step_lag()`, `step_sample()`, `step_shuffle()`, `step_slice()`

Examples

```
recipe(Ozone ~ ., data = airquality) %>%
  step_naomit(Solar.R) %>%
  prep(airquality, verbose = FALSE) %>%
  bake(new_data = NULL)
```

`step_nnmf` *Non-Negative Matrix Factorization Signal Extraction*

Description

`step_nnmf` creates a *specification* of a recipe step that will convert numeric data into one or more non-negative components.

[Deprecated]

Please use `step_nnmf_sparse()` instead of this step function.

Usage

```
step_nnmf(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  num_comp = 2,
  num_run = 30,
  options = list(),
  res = NULL,
  columns = NULL,
  prefix = "NNMF",
  seed = sample.int(10^5, 1),
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("nnmf")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.

`num_comp`	The number of components to retain as new predictors. If `num_comp` is greater than the number of columns or the number of possible components, a smaller value will be used. If `num_comp = 0` is set then no transformation is done and selected variables will stay unchanged.
`num_run`	A positive integer for the number of computations runs used to obtain a consensus projection.
`options`	A list of options to `nmf()` in the NMF package by way of the `NNMF()` function in the `dimRed` package. **Note** that the arguments `data` and `ndim` should not be passed here, and that NMF's parallel processing is turned off in favor of resample-level parallelization.
`res`	The `NNMF()` object is stored here once this preprocessing step has been trained by `prep()`.
`columns`	A character string of variable names that will be populated elsewhere.
`prefix`	A character string that will be the prefix to the resulting new variables. See notes below.
`seed`	An integer that will be used to set the seed in isolation when computing the factorization.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Non-negative matrix factorization computes latent components that have non-negative values and take into account that the original data have non-negative values.

The argument `num_comp` controls the number of components that will be retained (the original variables that are used to derive the components are removed from the data). The new components will have names that begin with `prefix` and a sequence of numbers. The variable names are padded with zeros. For example, if `num < 10`, their names will be `NNMF1` - `NNMF9`. If `num = 101`, the names would be `NNMF001` - `NNMF101`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the selectors or variables selected) and the number of components is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_pca()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Examples

```
data(biomass, package = "modeldata")

# rec <- recipe(HHV ~ ., data = biomass) %>%
#   update_role(sample, new_role = "id var") %>%
#   update_role(dataset, new_role = "split variable") %>%
#   step_nnmf(all_numeric_predictors(), num_comp = 2, seed = 473, num_run = 2) %>%
#   prep(training = biomass)
#
# bake(rec, new_data = NULL)
#
# library(ggplot2)
# bake(rec, new_data = NULL) %>%
#  ggplot(aes(x = NNMF2, y = NNMF1, col = HHV)) + geom_point()
```

`step_nnmf_sparse` *Non-Negative Matrix Factorization Signal Extraction with lasso Penalization*

Description

`step_nnmf_sparse()` creates a *specification* of a recipe step that will convert numeric data into one or more non-negative components.

Usage

```
step_nnmf_sparse(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  num_comp = 2,
  penalty = 0.001,
  options = list(),
  res = NULL,
  prefix = "NNMF",
```

```
    seed = sample.int(10^5, 1),
    keep_original_cols = FALSE,
    skip = FALSE,
    id = rand_id("nnmf_sparse")
  )
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`num_comp`	The number of components to retain as new predictors. If `num_comp` is greater than the number of columns or the number of possible components, a smaller value will be used. If `num_comp = 0` is set then no transformation is done and selected variables will stay unchanged.
`penalty`	A non-negative number used as a penalization factor for the loadings. Values are usually between zero and one.
`options`	A list of options to `nmf()` in the RcppML package. That package has a separate function `setRcppMLthreads()` that controls the amount of internal parallelization. **Note** that the argument `A`, `k`, `L1`, and `seed` should not be passed here.
`res`	A matrix of loadings is stored here, along with the names of the original predictors, once this preprocessing step has been trained by `prep()`.
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`seed`	An integer that will be used to set the seed in isolation when computing the factorization.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Non-negative matrix factorization computes latent components that have non-negative values and take into account that the original data have non-negative values.

The argument `num_comp` controls the number of components that will be retained (the original variables that are used to derive the components are removed from the data). The new components will have names that begin with `prefix` and a sequence of numbers. The variable names are padded with zeros. For example, if `num < 10`, their names will be `NNMF1` - `NNMF9`. If `num = 101`, the names would be `NNMF001` - `NNMF101`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the selectors or variables selected) and the number of components is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Examples

```
library(Matrix)
data(biomass, package = "modeldata")

rec <- recipe(HHV ~ ., data = biomass) %>%
  update_role(sample, new_role = "id var") %>%
  update_role(dataset, new_role = "split variable") %>%
  step_nnmf_sparse(
    all_numeric_predictors(),
    num_comp = 2,
    seed = 473,
    penalty = 0.01
  ) %>%
  prep(training = biomass)

bake(rec, new_data = NULL)
#'
library(ggplot2)
bake(rec, new_data = NULL) %>%
  ggplot(aes(x = NNMF2, y = NNMF1, col = HHV)) +
  geom_point()
```

`step_normalize` *Center and scale numeric data*

Description

`step_normalize` creates a *specification* of a recipe step that will normalize numeric data to have a standard deviation of one and a mean of zero.

Usage

```
step_normalize(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  means = NULL,
  sds = NULL,
  na_rm = TRUE,
  skip = FALSE,
  id = rand_id("normalize")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`means`	A named numeric vector of means. This is `NULL` until computed by `prep()`.
`sds`	A named numeric vector of standard deviations This is `NULL` until computed by `prep()`.
`na_rm`	A logical value indicating whether `NA` values should be removed when computing the standard deviation and mean.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Centering data means that the average of a variable is subtracted from the data. Scaling data means that the standard deviation of a variable is divided out of the data. `step_normalize` estimates the variable standard deviations and means from the data used in the `training` argument of `prep.recipe`. `bake.recipe` then applies the scaling to new data sets using these estimates.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected), `value` (the standard deviations and means), and `statistic` for the type of value is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

Other normalization steps: `step_center()`, `step_range()`, `step_scale()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

norm_trans <- rec %>%
  step_normalize(carbon, hydrogen)

norm_obj <- prep(norm_trans, training = biomass_tr)

transformed_te <- bake(norm_obj, biomass_te)

biomass_te[1:10, names(transformed_te)]
transformed_te
tidy(norm_trans, number = 1)
tidy(norm_obj, number = 1)

# To keep the original variables in the output, use `step_mutate_at`:
```

```
norm_keep_orig <- rec %>%
  step_mutate_at(all_numeric_predictors(), fn = list(orig = ~.)) %>%
  step_normalize(-contains("orig"), -all_outcomes())

keep_orig_obj <- prep(norm_keep_orig, training = biomass_tr)
keep_orig_te <- bake(keep_orig_obj, biomass_te)
keep_orig_te
```

`step_novel` *Simple Value Assignments for Novel Factor Levels*

Description

`step_novel` creates a *specification* of a recipe step that will assign a previously unseen factor level to a new value.

Usage

```
step_novel(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  new_level = "new",
  objects = NULL,
  skip = FALSE,
  id = rand_id("novel")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`new_level`	A single character value that will be assigned to new factor levels.
`objects`	A list of objects that contain the information on factor levels that will be determined by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The selected variables are adjusted to have a new level (given by `new_level`) that is placed in the last position. During preparation there will be no data points associated with this new level since all of the data have been seen.

Note that if the original columns are character, they will be converted to factors by this step.

Missing values will remain missing.

If `new_level` is already in the data given to `prep`, an error is thrown.

When fitting a model that can deal with new factor levels, consider using `workflows::add_recipe()` with `allow_novel_levels = TRUE` set in `hardhat::default_recipe_blueprint()`. This will allow your model to handle new levels at prediction time, instead of throwing warnings or errors.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected) and `value` (the factor levels that is used for the new value) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

`dummy_names()`

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(Sacramento, package = "modeldata")

sacr_tr <- Sacramento[1:800, ]
sacr_te <- Sacramento[801:806, ]
sacr_te$city[3] <- "beeptown"
sacr_te$city[4] <- "boopville"

rec <- recipe(~ city + zip, data = sacr_tr)

rec <- rec %>%
  step_novel(city, zip)
rec <- prep(rec, training = sacr_tr)
```

```
processed <- bake(rec, sacr_te)
tibble(old = sacr_te$city, new = processed$city)

tidy(rec, number = 1)
```

`step_ns` *Natural Spline Basis Functions*

Description

`step_ns` creates a *specification* of a recipe step that will create new columns that are basis expansions of variables using natural splines.

Usage

```
step_ns(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  objects = NULL,
  deg_free = 2,
  options = list(),
  skip = FALSE,
  id = rand_id("ns")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`objects`	A list of `splines::ns()` objects created once the step has been trained.
`deg_free`	The degrees of freedom for the natural spline. As the degrees of freedom for a natural spline increase, more flexible and complex curves can be generated. When a single degree of freedom is used, the result is a rescaled version of the original data.
`options`	A list of options for `splines::ns()` which should not include `x` or `df`.

`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_ns` can create new features from a single variable that enable fitting routines to model this variable in a nonlinear manner. The extent of the possible nonlinearity is determined by the `df` or `knot` arguments of `splines::ns()`. The original variables are removed from the data and new columns are added. The naming convention for the new variables is `varname_ns_1` and so on.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be affected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

with_splines <- rec %>%
  step_ns(carbon, hydrogen)
with_splines <- prep(with_splines, training = biomass_tr)

expanded <- bake(with_splines, biomass_te)
```

`expanded`

`step_num2factor` *Convert Numbers to Factors*

Description

`step_num2factor` will convert one or more numeric vectors to factors (ordered or unordered). This can be useful when categories are encoded as integers.

Usage

```
step_num2factor(
  recipe,
  ...,
  role = NA,
  transform = function(x) x,
  trained = FALSE,
  levels,
  ordered = FALSE,
  skip = FALSE,
  id = rand_id("num2factor")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`transform`	A function taking a single argument `x` that can be used to modify the numeric values prior to determining the levels (perhaps using `base::as.integer()`). The output of a function should be an integer that corresponds to the value of `levels` that should be assigned. If not an integer, the value will be converted to an integer during `bake()`.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`levels`	A character vector of values that will be used as the levels. These are the numeric data converted to character and ordered. This is modified once `prep()` is executed.
`ordered`	A single logical value; should the factor(s) be ordered?
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.

`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `ordered` is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
library(dplyr)
data(attrition, package = "modeldata")

attrition %>%
  group_by(StockOptionLevel) %>%
  count()

amnt <- c("nothin", "meh", "some", "copious")

rec <-
  recipe(Attrition ~ StockOptionLevel, data = attrition) %>%
  step_num2factor(
    StockOptionLevel,
    transform = function(x) x + 1,
    levels = amnt
  )

encoded <- rec %>%
  prep() %>%
  bake(new_data = NULL)

table(encoded$StockOptionLevel, attrition$StockOptionLevel)


# an example for binning
```

```
binner <- function(x) {
  x <- cut(x, breaks = 1000 * c(0, 5, 10, 20), include.lowest = TRUE)
  # now return the group number
  as.numeric(x)
}

inc <- c("low", "med", "high")

rec <-
  recipe(Attrition ~ MonthlyIncome, data = attrition) %>%
  step_num2factor(
    MonthlyIncome,
    transform = binner,
    levels = inc,
    ordered = TRUE
  ) %>%
  prep()

encoded <- bake(rec, new_data = NULL)

table(encoded$MonthlyIncome, binner(attrition$MonthlyIncome))

# What happens when a value is out of range?
ceo <- attrition %>%
  slice(1) %>%
  mutate(MonthlyIncome = 10^10)

bake(rec, ceo)
```

`step_nzv` *Near-Zero Variance Filter*

Description

`step_nzv` creates a *specification* of a recipe step that will potentially remove variables that are highly sparse and unbalanced.

Usage

```
step_nzv(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  freq_cut = 95/5,
  unique_cut = 10,
  options = list(freq_cut = 95/5, unique_cut = 10),
  removals = NULL,
```

```
  skip = FALSE,
  id = rand_id("nzv")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`freq_cut, unique_cut`	Numeric parameters for the filtering process. See the Details section below.
`options`	A list of options for the filter (see Details below).
`removals`	A character string that contains the names of columns that should be removed. These values are not determined until `prep()` is called.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This step can potentially remove columns from the data set. This may cause issues for subsequent steps in your recipe if the missing columns are specifically referenced by name. To avoid this, see the advice in the *Tips for saving recipes and filtering columns* section of selections.

This step diagnoses predictors that have one unique value (i.e. are zero variance predictors) or predictors that have both of the following characteristics:

1. they have very few unique values relative to the number of samples and
2. the ratio of the frequency of the most common value to the frequency of the second most common value is large.

For example, an example of near-zero variance predictor is one that, for 1000 samples, has two distinct values and 999 of them are a single value.

To be flagged, first, the frequency of the most prevalent value over the second most frequent value (called the "frequency ratio") must be above `freq_cut`. Secondly, the "percent of unique values," the number of unique values divided by the total number of samples (times 100), must also be below `unique_cut`.

In the above example, the frequency ratio is 999 and the unique value percent is 0.2%.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be removed) is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

Other variable filter steps: `step_corr()`, `step_filter_missing()`, `step_lincomb()`, `step_rm()`, `step_select()`, `step_zv()`

Examples

```
data(biomass, package = "modeldata")

biomass$sparse <- c(1, rep(0, nrow(biomass) - 1))

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(HHV ~ carbon + hydrogen + oxygen +
  nitrogen + sulfur + sparse,
data = biomass_tr
)

nzv_filter <- rec %>%
  step_nzv(all_predictors())

filter_obj <- prep(nzv_filter, training = biomass_tr)

filtered_te <- bake(filter_obj, biomass_te)
any(names(filtered_te) == "sparse")

tidy(nzv_filter, number = 1)
tidy(filter_obj, number = 1)
```

`step_ordinalscore` *Convert Ordinal Factors to Numeric Scores*

Description

`step_ordinalscore` creates a *specification* of a recipe step that will convert ordinal factor variables into numeric scores.

Usage

```
step_ordinalscore(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = NULL,
  convert = as.numeric,
  skip = FALSE,
  id = rand_id("ordinalscore")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character string of variables that will be converted. This is `NULL` until computed by `prep()`.
`convert`	A function that takes an ordinal factor vector as an input and outputs a single numeric variable.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Dummy variables from ordered factors with `C` levels will create polynomial basis functions with `C-1` terms. As an alternative, this step can be used to translate the ordered levels into a single numeric vector of values that represent (subjective) scores. By default, the translation uses a linear scale (1, 2, 3, ... `C`) but custom score functions can also be used (see the example below).

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be affected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
fail_lvls <- c("meh", "annoying", "really_bad")

ord_data <-
  data.frame(
    item = c("paperclip", "twitter", "airbag"),
    fail_severity = factor(fail_lvls,
      levels = fail_lvls,
      ordered = TRUE
    )
  )

model.matrix(~fail_severity, data = ord_data)

linear_values <- recipe(~ item + fail_severity, data = ord_data) %>%
  step_dummy(item) %>%
  step_ordinalscore(fail_severity)

linear_values <- prep(linear_values, training = ord_data)

bake(linear_values, new_data = NULL, everything())

custom <- function(x) {
  new_values <- c(1, 3, 7)
  new_values[as.numeric(x)]
}

nonlin_scores <- recipe(~ item + fail_severity, data = ord_data) %>%
  step_dummy(item) %>%
  step_ordinalscore(fail_severity, convert = custom)

tidy(nonlin_scores, number = 2)

nonlin_scores <- prep(nonlin_scores, training = ord_data)

bake(nonlin_scores, new_data = NULL, everything())

tidy(nonlin_scores, number = 2)
```

`step_other` *Collapse Some Categorical Levels*

Description

`step_other` creates a *specification* of a recipe step that will potentially pool infrequently occurring values into an "other" category.

Usage

```
step_other(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  threshold = 0.05,
  other = "other",
  objects = NULL,
  skip = FALSE,
  id = rand_id("other")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`threshold`	A numeric value between 0 and 1, or an integer greater or equal to one. If less than one, then factor levels with a rate of occurrence in the training set below `threshold` will be pooled to `other`. If greater or equal to one, then this value is treated as a frequency and factor levels that occur less than `threshold` times will be pooled to `other`.
`other`	A single character value for the "other" category.
`objects`	A list of objects that contain the information to pool infrequent levels that is determined by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The overall proportion (or total counts) of the categories are computed. The "other" category is used in place of any categorical levels whose individual proportion (or frequency) in the training set is less than `threshold`.

If no pooling is done the data are unmodified (although character data may be changed to factors based on the value of `strings_as_factors` in `prep()`). Otherwise, a factor is always returned with different factor levels.

If `threshold` is less than the largest category proportion, all levels except for the most frequent are collapsed to the `other` level.

If the retained categories include the value of `other`, an error is thrown. If `other` is in the list of discarded levels, no error occurs.

If no pooling is done, novel factor levels are converted to missing. If pooling is needed, they will be placed into the other category.

When data to be processed contains novel levels (i.e., not contained in the training set), the other category is assigned.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected) and `retained` (the factor levels that were not pulled into "other") is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

`dummy_names()`

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(Sacramento, package = "modeldata")

set.seed(19)
in_train <- sample(1:nrow(Sacramento), size = 800)
```

```
sacr_tr <- Sacramento[in_train, ]
sacr_te <- Sacramento[-in_train, ]

rec <- recipe(~ city + zip, data = sacr_tr)


rec <- rec %>%
  step_other(city, zip, threshold = .1, other = "other values")
rec <- prep(rec, training = sacr_tr)

collapsed <- bake(rec, sacr_te)
table(sacr_te$city, collapsed$city, useNA = "always")

tidy(rec, number = 1)

# novel levels are also "othered"
tahiti <- Sacramento[1, ]
tahiti$zip <- "a magical place"
bake(rec, tahiti)

# threshold as a frequency
rec <- recipe(~ city + zip, data = sacr_tr)

rec <- rec %>%
  step_other(city, zip, threshold = 2000, other = "other values")
rec <- prep(rec, training = sacr_tr)

tidy(rec, number = 1)
# compare it to
# sacr_tr %>% count(city, sort = TRUE) %>% top_n(4)
# sacr_tr %>% count(zip, sort = TRUE) %>% top_n(3)
```

`step_pca` *PCA Signal Extraction*

Description

`step_pca` creates a *specification* of a recipe step that will convert numeric data into one or more principal components.

Usage

```
step_pca(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  num_comp = 5,
  threshold = NA,
```

```
  options = list(),
  res = NULL,
  columns = NULL,
  prefix = "PC",
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("pca")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`num_comp`	The number of components to retain as new predictors. If `num_comp` is greater than the number of columns or the number of possible components, a smaller value will be used. If `num_comp = 0` is set then no transformation is done and selected variables will stay unchanged.
`threshold`	A fraction of the total variance that should be covered by the components. For example, `threshold = .75` means that `step_pca` should generate enough components to capture 75 percent of the variability in the variables. Note: using this argument will override and reset any value given to `num_comp`.
`options`	A list of options to the default method for `stats::prcomp()`. Argument defaults are set to `retx = FALSE`, `center = FALSE`, `scale. = FALSE`, and `tol = NULL`. **Note** that the argument `x` should not be passed here (or at all).
`res`	The `stats::prcomp.default()` object is stored here once this preprocessing step has be trained by `prep()`.
`columns`	A character string of variable names that will be populated elsewhere.
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Principal component analysis (PCA) is a transformation of a group of variables that produces a new set of artificial features or components. These components are designed to capture the maximum amount of information (i.e. variance) in the original variables. Also, the components are statistically independent from one another. This means that they can be used to combat large inter-variables correlations in a data set.

It is advisable to standardize the variables prior to running PCA. Here, each variable will be centered and scaled prior to the PCA calculation. This can be changed using the `options` argument or by using `step_center()` and `step_scale()`.

The argument `num_comp` controls the number of components that will be retained (the original variables that are used to derive the components are removed from the data). The new components will have names that begin with `prefix` and a sequence of numbers. The variable names are padded with zeros. For example, if `num_comp < 10`, their names will be `PC1 - PC9`. If `num_comp = 101`, the names would be `PC001 - PC101`.

Alternatively, `threshold` can be used to determine the number of components that are required to capture a specified fraction of the total variance in the variables.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, use either `type = "coef"` for the variable loadings per component or `type = "variance"` for how much variance each component accounts for.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

References

Jolliffe, I. T. (2010). *Principal Component Analysis*. Springer.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pls()`, `step_ratio()`, `step_spatialsign()`

Examples

```
rec <- recipe(~., data = USArrests)
pca_trans <- rec %>%
  step_normalize(all_numeric()) %>%
  step_pca(all_numeric(), num_comp = 3)
pca_estimates <- prep(pca_trans, training = USArrests)
```

```
pca_data <- bake(pca_estimates, USArrests)

rng <- extendrange(c(pca_data$PC1, pca_data$PC2))
plot(pca_data$PC1, pca_data$PC2,
  xlim = rng, ylim = rng
)

with_thresh <- rec %>%
  step_normalize(all_numeric()) %>%
  step_pca(all_numeric(), threshold = .99)
with_thresh <- prep(with_thresh, training = USArrests)
bake(with_thresh, USArrests)

tidy(pca_trans, number = 2)
tidy(pca_estimates, number = 2)
```

`step_percentile` *Percentile Transformation*

Description

`step_percentile` creates a *specification* of a recipe step that replaces the value of a variable with its percentile from the training set.

Usage

```
step_percentile(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  ref_dist = NULL,
  options = list(probs = (0:100)/100),
  skip = FALSE,
  id = rand_id("percentile")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.

`ref_dist`	The computed percentiles is stored here once this preprocessing step has be trained by `prep()`.
`options`	A named list of options to pass to `stats::quantile()`. See Details for more information.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
) %>%
  step_percentile(carbon)

prepped_rec <- prep(rec)

prepped_rec %>%
  bake(biomass_te)

tidy(rec, 1)
tidy(prepped_rec, 1)
```

`step_pls` *Partial Least Squares Feature Extraction*

Description

`step_pls` creates a *specification* of a recipe step that will convert numeric data into one or more new dimensions.

Usage

```
step_pls(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  num_comp = 2,
  predictor_prop = 1,
  outcome = NULL,
  options = list(scale = TRUE),
  preserve = deprecated(),
  res = NULL,
  columns = NULL,
  prefix = "PLS",
  keep_original_cols = FALSE,
  skip = FALSE,
  id = rand_id("pls")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`num_comp`	The number of components to retain as new predictors. If `num_comp` is greater than the number of columns or the number of possible components, a smaller value will be used. If `num_comp = 0` is set then no transformation is done and selected variables will stay unchanged.
`predictor_prop`	The maximum number of original predictors that can have non-zero coefficients for each PLS component (via regularization).
`outcome`	When a single outcome is available, character string or call to `dplyr::vars()` can be used to specify a single outcome variable.

`options`	A list of options to `mixOmics::pls()`, `mixOmics::spls()`, `mixOmics::plsda()`, or `mixOmics::splsda()` (depending on the data and arguments).
`preserve`	Use `keep_original_cols` instead to specify whether the original predictor data should be retained along with the new features.
`res`	A list of results are stored here once this preprocessing step has been trained by `prep()`.
`columns`	A character string of variable names that will be populated elsewhere.
`prefix`	A character string for the prefix of the resulting new variables. See notes below.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `FALSE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

PLS is a supervised version of principal component analysis that requires the outcome data to compute the new features.

This step requires the Bioconductor **mixOmics** package. If not installed, the step will stop with a note about installing the package.

The argument `num_comp` controls the number of components that will be retained (the original variables that are used to derive the components are removed from the data). The new components will have names that begin with `prefix` and a sequence of numbers. The variable names are padded with zeros. For example, if `num_comp < 10`, their names will be `PLS1 - PLS9`. If `num_comp = 101`, the names would be `PLS001 - PLS101`.

Sparsity can be encouraged using the `predictor_prop` parameter. This affects each PLS component, and indicates the maximum proportion of predictors with non-zero coefficients in each component. `step_pls()` converts this proportion to determine the `keepX` parameter in `mixOmics::spls()` and `mixOmics::splsda()`. See the references in `mixOmics::spls()` for details.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

The `tidy()` method returns the coefficients that are usually defined as

$$W(P'W)^{-1}$$

(See the Wikipedia article below)

When applied to data, these values are usually scaled by a column-specific norm. The `tidy()` method applies this same norm to the coefficients shown above. When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected), `components`, and `values` is returned.

Case weights

The underlying operation does not allow for case weights.

References

https://en.wikipedia.org/wiki/Partial_least_squares_regression

Rohart F, Gautier B, Singh A, Lê Cao K-A (2017) *mixOmics: An R package for 'omics feature selection and multiple data integration.* PLoS Comput Biol 13(11): e1005752. doi:10.1371/journal.pcbi.1005752

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_ratio()`, `step_spatialsign()`

Examples

```
# requires the Bioconductor mixOmics package
data(biomass, package = "modeldata")

biom_tr <-
  biomass %>%
  dplyr::filter(dataset == "Training") %>%
  dplyr::select(-dataset, -sample)
biom_te <-
  biomass %>%
  dplyr::filter(dataset == "Testing") %>%
  dplyr::select(-dataset, -sample, -HHV)

dense_pls <-
  recipe(HHV ~ ., data = biom_tr) %>%
  step_pls(all_numeric_predictors(), outcome = "HHV", num_comp = 3)

sparse_pls <-
  recipe(HHV ~ ., data = biom_tr) %>%
 step_pls(all_numeric_predictors(), outcome = "HHV", num_comp = 3, predictor_prop = 4 / 5)

## -----------------------------------------------------------------------------
## PLS discriminant analysis

data(cells, package = "modeldata")

cell_tr <-
  cells %>%
  dplyr::filter(case == "Train") %>%
  dplyr::select(-case)
cell_te <-
  cells %>%
  dplyr::filter(case == "Test") %>%
```

```
  dplyr::select(-case, -class)

dense_plsda <-
  recipe(class ~ ., data = cell_tr) %>%
  step_pls(all_numeric_predictors(), outcome = "class", num_comp = 5)

sparse_plsda <-
  recipe(class ~ ., data = cell_tr) %>%
 step_pls(all_numeric_predictors(), outcome = "class", num_comp = 5, predictor_prop = 1 / 4)
```

`step_poly` *Orthogonal Polynomial Basis Functions*

Description

`step_poly` creates a *specification* of a recipe step that will create new columns that are basis expansions of variables using orthogonal polynomials.

Usage

```
step_poly(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  objects = NULL,
  degree = 2,
  options = list(),
  skip = FALSE,
  id = rand_id("poly")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`objects`	A list of `stats::poly()` objects created once the step has been trained.
`degree`	The polynomial degree (an integer).

`options`	A list of options for `stats::poly()` which should not include `x`, `degree`, or `simple`. Note that the option `raw = TRUE` will produce the regular polynomial values (not orthogonalized).
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

`step_poly` can create new features from a single variable that enable fitting routines to model this variable in a nonlinear manner. The extent of the possible nonlinearity is determined by the `degree` argument of `stats::poly()`. The original variables are removed from the data and new columns are added. The naming convention for the new variables is `varname_poly_1` and so on.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected) and `degree` is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_relu()`, `step_sqrt()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

quadratic <- rec %>%
```

```
  step_poly(carbon, hydrogen)
quadratic <- prep(quadratic, training = biomass_tr)

expanded <- bake(quadratic, biomass_te)
expanded

tidy(quadratic, number = 1)
```

`step_profile` *Create a Profiling Version of a Data Set*

Description

`step_profile` creates a *specification* of a recipe step that will fix the levels of all variables but one and will create a sequence of values for the remaining variable. This step can be helpful when creating partial regression plots for additive models.

Usage

```
step_profile(
  recipe,
  ...,
  profile = NULL,
  pct = 0.5,
  index = 1,
  grid = list(pctl = TRUE, len = 100),
  columns = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = rand_id("profile")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`profile`	A call to `dplyr::vars()` to specify which variable will be profiled (see `selections()`). If a column is included in both lists to be fixed and to be profiled, an error is thrown.
`pct`	A value between 0 and 1 that is the percentile to fix continuous variables. This is applied to all continuous variables captured by the selectors. For date variables, either the minimum, median, or maximum used based on their distance to `pct`.

`index`	The level that qualitative variables will be fixed. If the variables are character (not factors), this will be the index of the sorted unique values. This is applied to all qualitative variables captured by the selectors.
`grid`	A named list with elements `pctl` (a logical) and `len` (an integer). If `pctl = TRUE`, then `len` denotes how many percentiles to use to create the profiling grid. This creates a grid between 0 and 1 and the profile is determined by the percentiles of the data. For example, if `pctl = TRUE` and `len = 3`, the profile would contain the minimum, median, and maximum values. If `pctl = FALSE`, it defines how many grid points between the minimum and maximum values should be created. This parameter is ignored for qualitative variables (since all of their possible levels are profiled). In the case of date variables, `pctl = FALSE` will always be used since there is no quantile method for dates.
`columns`	A character string that contains the names of columns that should be fixed and their values. These values are not determined until `prep()` is called.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This step is atypical in that, when baked, the `new_data` argument is ignored; the resulting data set is based on the fixed and profiled variable's information.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (which is the columns that will be affected) and `type` (fixed or profiled) is returned.

Case weights

The underlying operation does not allow for case weights.

Examples

```
data(Sacramento, package = "modeldata")

# Setup a grid across beds but keep the other values fixed
recipe(~ city + price + beds, data = Sacramento) %>%
```

```
  step_profile(-beds, profile = vars(beds)) %>%
  prep(training = Sacramento) %>%
  juice()


##########

# An *additive* model; not for use when there are interactions or
# other functional relationships between predictors

lin_mod <- lm(mpg ~ poly(disp, 2) + cyl + hp, data = mtcars)

# Show the difference in the two grid creation methods

disp_pctl <- recipe(~ disp + cyl + hp, data = mtcars) %>%
  step_profile(-disp, profile = vars(disp)) %>%
  prep(training = mtcars)

disp_grid <- recipe(~ disp + cyl + hp, data = mtcars) %>%
  step_profile(
    -disp,
    profile = vars(disp),
    grid = list(pctl = FALSE, len = 100)
  ) %>%
  prep(training = mtcars)

grid_data <- bake(disp_grid, new_data = NULL)
grid_data <- grid_data %>%
  mutate(
    pred = predict(lin_mod, grid_data),
    method = "grid"
  )

pctl_data <- bake(disp_pctl, new_data = NULL)
pctl_data <- pctl_data %>%
  mutate(
    pred = predict(lin_mod, pctl_data),
    method = "percentile"
  )

plot_data <- bind_rows(grid_data, pctl_data)

library(ggplot2)

ggplot(plot_data, aes(x = disp, y = pred)) +
  geom_point(alpha = .5, cex = 1) +
  facet_wrap(~method)
```

`step_range` *Scaling Numeric Data to a Specific Range*

Description

`step_range` creates a *specification* of a recipe step that will normalize numeric data to be within a pre-defined range of values.

Usage

```
step_range(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  min = 0,
  max = 1,
  ranges = NULL,
  skip = FALSE,
  id = rand_id("range")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`min`	A single numeric value for the smallest value in the range.
`max`	A single numeric value for the largest value in the range.
`ranges`	A character vector of variables that will be normalized. Note that this is ignored until the values are determined by `prep()`. Setting this value will be ineffective.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

When a new data point is outside of the ranges seen in the training set, the new values are truncated at `min` or `max`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected), `min`, and `max` is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other normalization steps: `step_center()`, `step_normalize()`, `step_scale()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

ranged_trans <- rec %>%
  step_range(carbon, hydrogen)

ranged_obj <- prep(ranged_trans, training = biomass_tr)

transformed_te <- bake(ranged_obj, biomass_te)

biomass_te[1:10, names(transformed_te)]
transformed_te

tidy(ranged_trans, number = 1)
tidy(ranged_obj, number = 1)
```

`step_ratio` *Ratio Variable Creation*

Description

`step_ratio` creates a *specification* of a recipe step that will create one or more ratios out of numeric variables.

Usage

```
step_ratio(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  denom = denom_vars(),
  naming = function(numer, denom) {
     make.names(paste(numer, denom, sep = "_o_"))

    },
  columns = NULL,
  keep_original_cols = TRUE,
  skip = FALSE,
  id = rand_id("ratio")
)

denom_vars(...)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose which variables will be used in the *numerator* of the ratio. When used with `denom_vars`, the dots indicate which variables are used in the *denominator*. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`denom`	A call to `denom_vars` to specify which variables are used in the denominator that can include specific variable names separated by commas or different selectors (see `selections()`). If a column is included in both lists to be numerator and denominator, it will be removed from the listing.
`naming`	A function that defines the naming convention for new ratio columns.
`columns`	The column names used in the ratios. This argument is not populated until `prep()` is executed.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `TRUE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `denom` is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_spatialsign()`

Examples

```
library(recipes)
data(biomass, package = "modeldata")

biomass$total <- apply(biomass[, 3:7], 1, sum)
biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(HHV ~ carbon + hydrogen + oxygen + nitrogen +
  sulfur + total,
data = biomass_tr
)

ratio_recipe <- rec %>%
  # all predictors over total
  step_ratio(all_numeric_predictors(), denom = denom_vars(total)) %>%
  # get rid of the original predictors
  step_rm(all_predictors(), -ends_with("total"))

ratio_recipe <- prep(ratio_recipe, training = biomass_tr)

ratio_data <- bake(ratio_recipe, biomass_te)
ratio_data
```

---

step_regex *Detect a regular expression*

Description

`step_regex` creates a *specification* of a recipe step that will create a new dummy variable based on a regular expression.

Usage

```
step_regex(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  pattern = ".",
  options = list(),
  result = make.names(pattern),
  input = NULL,
  skip = FALSE,
  id = rand_id("regex")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	A single selector function to choose which variable will be searched for the regex pattern. The selector should resolve to a single variable. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`pattern`	A character string containing a regular expression (or character string for `fixed = TRUE`) to be matched in the given character vector. Coerced by `as.character` to a character string if possible.
`options`	A list of options to `grepl()` that should not include `x` or `pattern`.
`result`	A single character value for the name of the new variable. It should be a valid column name.
`input`	A single character value for the name of the variable being searched. This is `NULL` until computed by `prep()`.

`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `result` (the new column name) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(covers, package = "modeldata")

rec <- recipe(~description, covers) %>%
  step_regex(description, pattern = "(rock|stony)", result = "rocks") %>%
  step_regex(description, pattern = "ratake families")

rec2 <- prep(rec, training = covers)
rec2

with_dummies <- bake(rec2, new_data = covers)
with_dummies
tidy(rec, number = 1)
tidy(rec2, number = 1)
```

`step_relevel` *Relevel factors to a desired level*

Description

`step_relevel` creates a *specification* of a recipe step that will reorder the provided factor columns so that the level specified by ref_level is first. This is useful for contr.treatment contrasts which take the first level as the reference.

Usage

```
step_relevel(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  ref_level,
  objects = NULL,
  skip = FALSE,
  id = rand_id("relevel")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`ref_level`	A single character value that will be used to relevel the factor column(s) (if the level is present).
`objects`	A list of objects that contain the information on factor levels that will be determined by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The selected variables are releveled to a level (given by `ref_level`). Placing the `ref_level` in the first position.

Note that if the original columns are character, they will be converted to factors by this step.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_string2factor()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(Sacramento, package = "modeldata")
rec <- recipe(~ city + zip, data = Sacramento) %>%
  step_unknown(city, new_level = "UNKNOWN") %>%
  step_relevel(city, ref_level = "UNKNOWN") %>%
  prep()

data <- bake(rec, Sacramento)
levels(data$city)
```

---

step_relu *Apply (Smoothed) Rectified Linear Transformation*

Description

`step_relu` creates a *specification* of a recipe step that will apply the rectified linear or softplus transformations to numeric data. The transformed data is added as new columns to the data matrix.

Usage

```
step_relu(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  shift = 0,
  reverse = FALSE,
  smooth = FALSE,
  prefix = "right_relu_",
  columns = NULL,
```

```
  skip = FALSE,
  id = rand_id("relu")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`shift`	A numeric value dictating a translation to apply to the data.
`reverse`	A logical to indicate if the left hinge should be used as opposed to the right hinge.
`smooth`	A logical indicating if the softplus function, a smooth approximation to the rectified linear transformation, should be used.
`prefix`	A prefix for generated column names, defaults to "right_relu_" for right hinge transformation and "left_relu_" for reversed/left hinge transformations.
`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The rectified linear transformation is calculated as

$$max(0, x - c)$$

and is also known as the ReLu or right hinge function. If `reverse` is true, then the transformation is reflected about the y-axis, like so:

$$max(0, c - x)$$

Setting the `smooth` option to true will instead calculate a smooth approximation to ReLu according to

$$ln(1 + e^(x - c)$$

The `reverse` argument may also be applied to this transformation.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Connection to MARS

The rectified linear transformation is used in Multivariate Adaptive Regression Splines as a basis function to fit piecewise linear functions to data in a strategy similar to that employed in tree based models. The transformation is a popular choice as an activation function in many neural networks, which could then be seen as a stacked generalization of MARS when making use of ReLu activations. The hinge function also appears in the loss function of Support Vector Machines, where it penalizes residuals only if they are within a certain margin of the decision boundary.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_sqrt()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

transformed_te <- rec %>%
  step_relu(carbon, shift = 40) %>%
  prep(biomass_tr) %>%
  bake(biomass_te)

transformed_te
```

`step_rename` *Rename variables by name using dplyr*

Description

`step_rename` creates a *specification* of a recipe step that will add variables using `dplyr::rename()`.

Usage

```
step_rename(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  inputs = NULL,
  skip = FALSE,
  id = rand_id("rename")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more unquoted expressions separated by commas. See `dplyr::rename()` where the convention is `new_name = old_name`.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`inputs`	Quosure(s) of `...`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

When an object in the user's global environment is referenced in the expression defining the new variable(s), it is a good idea to use quasiquotation (e.g. `!!`) to embed the value of the object in the expression (to be portable between sessions).

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `values` which contains the `rename` expressions as character strings (and are not reparsable) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dplyr steps: `step_arrange()`, `step_filter()`, `step_mutate_at()`, `step_mutate()`, `step_rename_at()`, `step_sample()`, `step_select()`, `step_slice()`

Examples

```
recipe(~., data = iris) %>%
  step_rename(Sepal_Width = Sepal.Width) %>%
  prep() %>%
  bake(new_data = NULL) %>%
  slice(1:5)

vars <- c(var1 = "cyl", var2 = "am")
car_rec <-
  recipe(~., data = mtcars) %>%
  step_rename(!!vars)

car_rec %>%
  prep() %>%
  bake(new_data = NULL)

car_rec %>%
  tidy(number = 1)
```

`step_rename_at` *Rename multiple columns using dplyr*

Description

`step_rename_at` creates a *specification* of a recipe step that will rename the selected variables using a common function via `dplyr::rename_at()`.

Usage

```
step_rename_at(
  recipe,
  ...,
  fn,
  role = "predictor",
  trained = FALSE,
  inputs = NULL,
  skip = FALSE,
  id = rand_id("rename_at")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`fn`	A function `fun`, a quosure style lambda ‘~ fun(.)“ or a list of either form (but containing only a single function, see `dplyr::rename_at()`). **Note that this argument must be named**.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`inputs`	A vector of column names populated by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` which contains the columns being transformed is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dplyr steps: `step_arrange()`, `step_filter()`, `step_mutate_at()`, `step_mutate()`, `step_rename()`, `step_sample()`, `step_select()`, `step_slice()`

Examples

```
library(dplyr)
recipe(~., data = iris) %>%
  step_rename_at(everything(), fn = ~ gsub(".", "_", ., fixed = TRUE)) %>%
  prep() %>%
  bake(new_data = NULL) %>%
  slice(1:10)
```

`step_rm` *General Variable Filter*

Description

`step_rm` creates a *specification* of a recipe step that will remove variables based on their name, type, or role.

Usage

```
step_rm(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  removals = NULL,
  skip = FALSE,
  id = rand_id("rm")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`removals`	A character string that contains the names of columns that should be removed. These values are not determined until `prep()` is called.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This step can potentially remove columns from the data set. This may cause issues for subsequent steps in your recipe if the missing columns are specifically referenced by name. To avoid this, see the advice in the *Tips for saving recipes and filtering columns* section of selections.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be removed) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other variable filter steps: `step_corr()`, `step_filter_missing()`, `step_lincomb()`, `step_nzv()`, `step_select()`, `step_zv()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

library(dplyr)
smaller_set <- rec %>%
  step_rm(contains("gen"))

smaller_set <- prep(smaller_set, training = biomass_tr)

filtered_te <- bake(smaller_set, biomass_te)
filtered_te

tidy(smaller_set, number = 1)
```

`step_sample` *Sample rows using dplyr*

Description

`step_sample` creates a *specification* of a recipe step that will sample rows using `dplyr::sample_n()` or `dplyr::sample_frac()`.

Usage

```
step_sample(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  size = NULL,
  replace = FALSE,
  skip = TRUE,
  id = rand_id("sample")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	Argument ignored; included for consistency with other step specification functions.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`size`	An integer or fraction. If the value is within (0, 1), `dplyr::sample_frac()` is applied to the data. If an integer value of 1 or greater is used, `dplyr::sample_n()` is applied. The default of `NULL` uses `dplyr::sample_n()` with the size of the training set (or smaller for smaller `new_data`).
`replace`	Sample with or without replacement?
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = FALSE`.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Row Filtering

This step can entirely remove observations (rows of data), which can have unintended and/or problematic consequences when applying the step to new data later via `bake()`. Consider whether `skip = TRUE` or `skip = FALSE` is more appropriate in any given use case. In most instances that affect the rows of the data being predicted, this step probably should not be applied at all; instead, execute operations like this outside and before starting a preprocessing `recipe()`.

Tidying

When you `tidy()` this step, a tibble with columns `size`, `replace`, and `id` is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

See Also

Other row operation steps: `step_arrange()`, `step_filter()`, `step_impute_roll()`, `step_lag()`, `step_naomit()`, `step_shuffle()`, `step_slice()`

Other dplyr steps: `step_arrange()`, `step_filter()`, `step_mutate_at()`, `step_mutate()`, `step_rename_at()`, `step_rename()`, `step_select()`, `step_slice()`

Examples

```
# Uses `sample_n`
recipe(~., data = mtcars) %>%
  step_sample(size = 1) %>%
  prep(training = mtcars) %>%
  bake(new_data = NULL) %>%
  nrow()

# Uses `sample_frac`
recipe(~., data = mtcars) %>%
  step_sample(size = 0.9999) %>%
  prep(training = mtcars) %>%
  bake(new_data = NULL) %>%
  nrow()

# Uses `sample_n` and returns _at maximum_ 20 samples.
smaller_cars <-
  recipe(~., data = mtcars) %>%
  step_sample() %>%
  prep(training = mtcars %>% slice(1:20))

bake(smaller_cars, new_data = NULL) %>% nrow()
bake(smaller_cars, new_data = mtcars %>% slice(21:32)) %>% nrow()
```

`step_scale` *Scaling Numeric Data*

Description

`step_scale` creates a *specification* of a recipe step that will normalize numeric data to have a standard deviation of one.

Usage

```
step_scale(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  sds = NULL,
  factor = 1,
  na_rm = TRUE,
  skip = FALSE,
  id = rand_id("scale")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`sds`	A named numeric vector of standard deviations. This is `NULL` until computed by `prep()`.
`factor`	A numeric value of either 1 or 2 that scales the numeric inputs by one or two standard deviations. By dividing by two standard deviations, the coefficients attached to continuous predictors can be interpreted the same way as with binary inputs. Defaults to `1`. More in reference below.
`na_rm`	A logical value indicating whether `NA` values should be removed when computing the standard deviation.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Scaling data means that the standard deviation of a variable is divided out of the data. `step_scale` estimates the variable standard deviations from the data used in the `training` argument of `prep.recipe`. `bake.recipe` then applies the scaling to new data sets using these standard deviations.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `value` (the standard deviations) is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, case weights are only used with frequency weights. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

References

Gelman, A. (2007) "Scaling regression inputs by dividing by two standard deviations." Unpublished. Source: `http://www.stat.columbia.edu/~gelman/research/unpublished/standardizing.pdf`.

See Also

Other normalization steps: `step_center()`, `step_normalize()`, `step_range()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

scaled_trans <- rec %>%
  step_scale(carbon, hydrogen)

scaled_obj <- prep(scaled_trans, training = biomass_tr)

transformed_te <- bake(scaled_obj, biomass_te)

biomass_te[1:10, names(transformed_te)]
transformed_te
tidy(scaled_trans, number = 1)
tidy(scaled_obj, number = 1)
```

`step_select` *Select variables using dplyr*

Description

`step_select()` creates a *specification* of a recipe step that will select variables using `dplyr::select()`.

Usage

```
step_select(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = rand_id("select")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms selected by this step, what analysis role should they be assigned?
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

When an object in the user's global environment is referenced in the expression defining the new variable(s), it is a good idea to use quasiquotation (e.g. `!!`) to embed the value of the object in the expression (to be portable between sessions). See the examples.

This step can potentially remove columns from the data set. This may cause issues for subsequent steps in your recipe if the missing columns are specifically referenced by name. To avoid this, see the advice in the *Tips for saving recipes and filtering columns* section of selections.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` which contains the `select` expressions as character strings (and are not reparsable) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other variable filter steps: `step_corr()`, `step_filter_missing()`, `step_lincomb()`, `step_nzv()`, `step_rm()`, `step_zv()`

Other dplyr steps: `step_arrange()`, `step_filter()`, `step_mutate_at()`, `step_mutate()`, `step_rename_at()`, `step_rename()`, `step_sample()`, `step_slice()`

Examples

```
library(dplyr)

iris_tbl <- as_tibble(iris)
iris_train <- slice(iris_tbl, 1:75)
iris_test <- slice(iris_tbl, 76:150)

dplyr_train <- select(iris_train, Species, starts_with("Sepal"))
dplyr_test <- select(iris_test, Species, starts_with("Sepal"))

rec <- recipe(~., data = iris_train) %>%
  step_select(Species, starts_with("Sepal")) %>%
  prep(training = iris_train)

rec_train <- bake(rec, new_data = NULL)
all.equal(dplyr_train, rec_train)

rec_test <- bake(rec, iris_test)
all.equal(dplyr_test, rec_test)

# Local variables
sepal_vars <- c("Sepal.Width", "Sepal.Length")

qq_rec <-
  recipe(~., data = iris_train) %>%
  # fine for interactive usage
  step_select(Species, all_of(sepal_vars)) %>%
  # best approach for saving a recipe to disk
  step_select(Species, all_of(!!sepal_vars))

# Note that `sepal_vars` is inlined in the second approach
qq_rec
```

`step_shuffle` *Shuffle Variables*

Description

`step_shuffle` creates a *specification* of a recipe step that will randomly change the order of rows for selected variables.

Usage

```
step_shuffle(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("shuffle")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character string that contains the names of columns that should be shuffled. These values are not determined until `prep()` is called.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be permuted) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other row operation steps: `step_arrange()`, `step_filter()`, `step_impute_roll()`, `step_lag()`, `step_naomit()`, `step_sample()`, `step_slice()`

Examples

```
integers <- data.frame(A = 1:12, B = 13:24, C = 25:36)

library(dplyr)
rec <- recipe(~ A + B + C, data = integers) %>%
  step_shuffle(A, B)

rand_set <- prep(rec, training = integers)

set.seed(5377)
bake(rand_set, integers)

tidy(rec, number = 1)
tidy(rand_set, number = 1)
```

`step_slice` *Filter rows by position using dplyr*

Description

`step_slice` creates a *specification* of a recipe step that will filter rows using `dplyr::slice()`.

Usage

```
step_slice(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  inputs = NULL,
  skip = TRUE,
  id = rand_id("slice")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	Integer row values. See `dplyr::slice()` for more details.

`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`inputs`	Quosure of values given by `...`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = FALSE`.
`id`	A character string that is unique to this step to identify it.

Details

When an object in the user's global environment is referenced in the expression defining the new variable(s), it is a good idea to use quasiquotation (e.g. `!!`) to embed the value of the object in the expression (to be portable between sessions). See the examples.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Row Filtering

This step can entirely remove observations (rows of data), which can have unintended and/or problematic consequences when applying the step to new data later via `bake()`. Consider whether `skip = TRUE` or `skip = FALSE` is more appropriate in any given use case. In most instances that affect the rows of the data being predicted, this step probably should not be applied at all; instead, execute operations like this outside and before starting a preprocessing `recipe()`.

Tidying

When you `tidy()` this step, a tibble with column `terms` which contains the filtering indices is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other row operation steps: `step_arrange()`, `step_filter()`, `step_impute_roll()`, `step_lag()`, `step_naomit()`, `step_sample()`, `step_shuffle()`

Other dplyr steps: `step_arrange()`, `step_filter()`, `step_mutate_at()`, `step_mutate()`, `step_rename_at()`, `step_rename()`, `step_sample()`, `step_select()`

Examples

```
rec <- recipe(~., data = iris) %>%
  step_slice(1:3)

prepped <- prep(rec, training = iris %>% slice(1:75))
tidy(prepped, number = 1)

library(dplyr)

dplyr_train <-
  iris %>%
  as_tibble() %>%
  slice(1:75) %>%
  slice(1:3)

rec_train <- bake(prepped, new_data = NULL)
all.equal(dplyr_train, rec_train)

dplyr_test <-
  iris %>%
  as_tibble() %>%
  slice(76:150) %>%
  slice(1:3)
rec_test <- bake(prepped, iris %>% slice(76:150))
all.equal(dplyr_test, rec_test)

# Embedding the integer expression (or vector) into the
# recipe:

keep_rows <- 1:6

qq_rec <-
  recipe(~., data = iris) %>%
  # Embed `keep_rows` in the call using !!
  step_slice(!!keep_rows) %>%
  prep(training = iris)

tidy(qq_rec, number = 1)
```

`step_spatialsign` *Spatial Sign Preprocessing*

Description

`step_spatialsign` is a *specification* of a recipe step that will convert numeric data into a projection on to a unit sphere.

Usage

```
step_spatialsign(
  recipe,
  ...,
  role = "predictor",
  na_rm = TRUE,
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("spatialsign")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`na_rm`	A logical: should missing data be removed from the norm computation?
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The spatial sign transformation projects the variables onto a unit sphere and is related to global contrast normalization. The spatial sign of a vector `w` is `w/norm(w)`.

The variables should be centered and scaled prior to the computations.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be affected) is returned.

Case weights

This step performs an unsupervised operation that can utilize case weights. As a result, only frequency weights are allowed. For more information, see the documentation in case_weights and the examples on `tidymodels.org`.

Unlike most, this step requires the case weights to be available when new samples are processed (e.g., when `bake()` is used or `predict()` with a workflow). To tell recipes that the case weights are required at bake time, use `recipe %>% update_role_requirements(role = "case_weights", bake = TRUE)`. See `update_role_requirements()` for more information.

References

Serneels, S., De Nolf, E., and Van Espen, P. (2006). Spatial sign preprocessing: a simple way to impart moderate robustness to multivariate estimators. *Journal of Chemical Information and Modeling*, 46(3), 1402-1409.

See Also

Other multivariate transformation steps: `step_classdist()`, `step_depth()`, `step_geodist()`, `step_ica()`, `step_isomap()`, `step_kpca_poly()`, `step_kpca_rbf()`, `step_kpca()`, `step_mutate_at()`, `step_nnmf_sparse()`, `step_nnmf()`, `step_pca()`, `step_pls()`, `step_ratio()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

ss_trans <- rec %>%
  step_center(carbon, hydrogen) %>%
  step_scale(carbon, hydrogen) %>%
  step_spatialsign(carbon, hydrogen)

ss_obj <- prep(ss_trans, training = biomass_tr)

transformed_te <- bake(ss_obj, biomass_te)

plot(biomass_te$carbon, biomass_te$hydrogen)

plot(transformed_te$carbon, transformed_te$hydrogen)

tidy(ss_trans, number = 3)
tidy(ss_obj, number = 3)
```

`step_sqrt` *Square Root Transformation*

Description

`step_sqrt` creates a *specification* of a recipe step that will square root transform the data.

Usage

```
step_sqrt(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("sqrt")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be affected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_YeoJohnson()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`

Examples

```
set.seed(313)
examples <- matrix(rnorm(40)^2, ncol = 2)
examples <- as.data.frame(examples)

rec <- recipe(~ V1 + V2, data = examples)

sqrt_trans <- rec %>%
  step_sqrt(all_numeric_predictors())

sqrt_obj <- prep(sqrt_trans, training = examples)

transformed_te <- bake(sqrt_obj, examples)
plot(examples$V1, transformed_te$V1)

tidy(sqrt_trans, number = 1)
tidy(sqrt_obj, number = 1)
```

`step_string2factor` *Convert Strings to Factors*

Description

`step_string2factor` will convert one or more character vectors to factors (ordered or unordered).

Use this step only in special cases (see Details) and instead convert strings to factors before using any tidymodels functions.

Usage

```
step_string2factor(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  levels = NULL,
  ordered = FALSE,
  skip = FALSE,
  id = rand_id("string2factor")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`levels`	An options specification of the levels to be used for the new factor. If left `NULL`, the sorted unique values present when bake is called will be used.
`ordered`	A single logical value; should the factor(s) be ordered?
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

When should you use this step?:

In most cases, if you are planning to use `step_string2factor()` without setting `levels`, you will be better off converting those character variables to factor variables **before using a recipe**.

This can be done using **dplyr** with the following code

```
df <- mutate(df, across(where(is.character), as.factor))
```

During resampling, the complete set of values might not be in the character data. Converting them to factors with `step_string2factor()` then will misconfigure the levels.

If the `levels` argument to `step_string2factor()`is used, it will convert all variables affected by this step to have the same levels. Because of this, you will need to know the full set of level when you define the recipe.

Also, note that `prep()` has an option `strings_as_factors` that defaults to `TRUE`. This should be changed so that raw character data will be applied to `step_string2factor()`. However, this step can also take existing factors (but will leave them as-is).

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `ordered` is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_time()`, `step_unknown()`, `step_unorder()`

Examples

```
data(Sacramento, package = "modeldata")

# convert factor to string to demonstrate
Sacramento$city <- as.character(Sacramento$city)

rec <- recipe(~ city + zip, data = Sacramento)

make_factor <- rec %>%
  step_string2factor(city)

make_factor <- prep(make_factor,
  training = Sacramento
)

make_factor

# note that `city` is a factor in recipe output
bake(make_factor, new_data = NULL) %>% head()

# ...but remains a string in the data
Sacramento %>% head()
```

`step_time` *Time Feature Generator*

Description

`step_time()` creates a *specification* of a recipe step that will convert date-time data into one or more factor or numeric variables.

Usage

```
step_time(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  features = c("hour", "minute", "second"),
```

```
  columns = NULL,
  keep_original_cols = TRUE,
  skip = FALSE,
  id = rand_id("time")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. The selected variables should have class `POSIXct` or `POSIXlt`. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? By default, the new columns created by this step from the original variables will be used as *predictors* in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`features`	A character string that includes at least one of the following values: `am` (is is AM), `hour`, `hour12`, `minute`, `second`, `decimal_day`.
`columns`	A character string of variables that will be used as inputs. This field is a placeholder and will be populated once `prep()` is used.
`keep_original_cols`	A logical to keep the original variables in the output. Defaults to `TRUE`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Unlike some other steps, `step_time()` does *not* remove the original time variables by default. Set `keep_original_cols` to `FALSE` to remove them.

`decimal_day` return time of day as a decimal number between 0 and 24. for example `"07:15:00"` would be transformed to `7.25` and `"03:59:59"` would be transformed to `3.999722`. The formula for these calculations are `hour(x) + (second(x) + minute(x) * 60) / 3600`.

See `step_date()` if you want to calculate features that are larger than hours.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `value` (the feature names).

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_unknown()`, `step_unorder()`

Examples

```
library(lubridate)

examples <- data.frame(
  times = ymd_hms("2022-05-06 23:51:07") +
  hours(1:5) + minutes(1:5) + seconds(1:5)
)
time_rec <- recipe(~ times, examples) %>%
  step_time(all_predictors())

tidy(time_rec, number = 1)

time_rec <- prep(time_rec, training = examples)

time_values <- bake(time_rec, new_data = examples)
time_values

tidy(time_rec, number = 1)
```

`step_unknown` *Assign missing categories to "unknown"*

Description

step_unknown creates a *specification* of a recipe step that will assign a missing value in a factor level to"unknown".

Usage

```
step_unknown(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  new_level = "unknown",
  objects = NULL,
  skip = FALSE,
  id = rand_id("unknown")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`new_level`	A single character value that will be assigned to new factor levels.
`objects`	A list of objects that contain the information on factor levels that will be determined by `prep()`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The selected variables are adjusted to have a new level (given by `new_level`) that is placed in the last position.

Note that if the original columns are character, they will be converted to factors by this step.

If `new_level` is already in the data given to `prep`, an error is thrown.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the columns that will be affected) and `value` (the factor levels that is used for the new value) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

`dummy_names()`

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unorder()`

Examples

```
data(Sacramento, package = "modeldata")

rec <-
  recipe(~ city + zip, data = Sacramento) %>%
  step_unknown(city, new_level = "unknown city") %>%
  step_unknown(zip, new_level = "unknown zip") %>%
  prep()

table(bake(rec, new_data = NULL) %>% pull(city),
  Sacramento %>% pull(city),
  useNA = "always"
) %>%
  as.data.frame() %>%
  dplyr::filter(Freq > 0)

tidy(rec, number = 1)
```

`step_unorder` *Convert Ordered Factors to Unordered Factors*

Description

`step_unorder` creates a *specification* of a recipe step that will transform the data.

Usage

```
step_unorder(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("unorder")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.

`columns`	A character string of variable names that will be populated (eventually) by the `terms` argument.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The factors level order is preserved during the transformation.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be affected) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other dummy variable and encoding steps: `step_bin2factor()`, `step_count()`, `step_date()`, `step_dummy_extract()`, `step_dummy_multi_choice()`, `step_dummy()`, `step_factor2string()`, `step_holiday()`, `step_indicate_na()`, `step_integer()`, `step_novel()`, `step_num2factor()`, `step_ordinalscore()`, `step_other()`, `step_regex()`, `step_relevel()`, `step_string2factor()`, `step_time()`, `step_unknown()`

Examples

```
lmh <- c("Low", "Med", "High")

examples <- data.frame(
  X1 = factor(rep(letters[1:4], each = 3)),
  X2 = ordered(rep(lmh, each = 4),
    levels = lmh
  )
)

rec <- recipe(~ X1 + X2, data = examples)

factor_trans <- rec %>%
  step_unorder(all_nominal_predictors())
```

```
factor_obj <- prep(factor_trans, training = examples)

transformed_te <- bake(factor_obj, examples)
table(transformed_te$X2, examples$X2)

tidy(factor_trans, number = 1)
tidy(factor_obj, number = 1)
```

`step_window` *Moving Window Functions*

Description

`step_window` creates a *specification* of a recipe step that will create new columns that are the results of functions that compute statistics across moving windows.

Usage

```
step_window(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  size = 3,
  na_rm = TRUE,
  statistic = "mean",
  columns = NULL,
  names = NULL,
  skip = FALSE,
  id = rand_id("window")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	For model terms created by this step, what analysis role should they be assigned? If `names` is left to be `NULL`, the rolling statistics replace the original columns and the roles are left unchanged. If `names` is set, those new columns will have a role of `NULL` unless this argument has a value.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`size`	An odd integer `>= 3` for the window size.
`na_rm`	A logical for whether missing values should be removed from the calculations within each window.

`statistic`	A character string for the type of statistic that should be calculated for each moving window. Possible values are: `'max'`, `'mean'`, `'median'`, `'min'`, `'prod'`, `'sd'`, `'sum'`, `'var'`
`columns`	A character string that contains the names of columns that should be processed. These values are not determined until `prep()` is called.
`names`	An optional character string that is the same length of the number of terms selected by `terms`. If you are not sure what columns will be selected, use the `summary` function (see the example below). These will be the names of the new columns created by the step.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The calculations use a somewhat atypical method for handling the beginning and end parts of the rolling statistics. The process starts with the center justified window calculations and the beginning and ending parts of the rolling values are determined using the first and last rolling values, respectively. For example, if a column `x` with 12 values is smoothed with a 5-point moving median, the first three smoothed values are estimated by `median(x[1:5])` and the fourth uses `median(x[2:6])`.

step will stop with a note about installing the package.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected), `statistic` (the summary function name), and `size` is returned.

Case weights

The underlying operation does not allow for case weights.

Examples

```
library(recipes)
library(dplyr)
library(rlang)
library(ggplot2, quietly = TRUE)

set.seed(5522)
sim_dat <- data.frame(x1 = (20:100) / 10)
n <- nrow(sim_dat)
```

```
sim_dat$y1 <- sin(sim_dat$x1) + rnorm(n, sd = 0.1)
sim_dat$y2 <- cos(sim_dat$x1) + rnorm(n, sd = 0.1)
sim_dat$x2 <- runif(n)
sim_dat$x3 <- rnorm(n)

rec <- recipe(y1 + y2 ~ x1 + x2 + x3, data = sim_dat) %>%
  step_window(starts_with("y"),
    size = 7, statistic = "median",
    names = paste0("med_7pt_", 1:2),
    role = "outcome"
  ) %>%
  step_window(starts_with("y"),
    names = paste0("mean_3pt_", 1:2),
    role = "outcome"
  )
rec <- prep(rec, training = sim_dat)

smoothed_dat <- bake(rec, sim_dat, everything())

ggplot(data = sim_dat, aes(x = x1, y = y1)) +
  geom_point() +
  geom_line(data = smoothed_dat, aes(y = med_7pt_1)) +
  geom_line(data = smoothed_dat, aes(y = mean_3pt_1), col = "red") +
  theme_bw()

tidy(rec, number = 1)
tidy(rec, number = 2)

# If you want to replace the selected variables with the rolling statistic
# don't set `names`
sim_dat$original <- sim_dat$y1
rec <- recipe(y1 + y2 + original ~ x1 + x2 + x3, data = sim_dat) %>%
  step_window(starts_with("y"))
rec <- prep(rec, training = sim_dat)
smoothed_dat <- bake(rec, sim_dat, everything())
ggplot(smoothed_dat, aes(x = original, y = y1)) +
  geom_point() +
  theme_bw()
```

`step_YeoJohnson` *Yeo-Johnson Transformation*

Description

`step_YeoJohnson` creates a *specification* of a recipe step that will transform data using a simple Yeo-Johnson transformation.

Usage

```
step_YeoJohnson(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  lambdas = NULL,
  limits = c(-5, 5),
  num_unique = 5,
  na_rm = TRUE,
  skip = FALSE,
  id = rand_id("YeoJohnson")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`lambdas`	A numeric vector of transformation values. This is `NULL` until computed by `prep()`.
`limits`	A length 2 numeric vector defining the range to compute the transformation parameter lambda.
`num_unique`	An integer where data that have less possible values will not be evaluated for a transformation.
`na_rm`	A logical value indicating whether `NA` values should be removed during computations.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The Yeo-Johnson transformation is very similar to the Box-Cox but does not require the input variables to be strictly positive. In the package, the partial log-likelihood function is directly optimized within a reasonable set of transformation values (which can be changed by the user).

This transformation is typically done on the outcome variable using the residuals for a statistical model (such as ordinary least squares). Here, a simple null model (intercept only) is used to apply the transformation to the *predictor* variables individually. This can have the effect of making the variable distributions more symmetric.

If the transformation parameters are estimated to be very closed to the bounds, or if the optimization fails, a value of `NA` is used and no transformation is applied.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (the selectors or variables selected) and `value` (the lambda estimate) is returned.

Case weights

The underlying operation does not allow for case weights.

References

Yeo, I. K., and Johnson, R. A. (2000). A new family of power transformations to improve normality or symmetry. *Biometrika*.

See Also

Other individual transformation steps: `step_BoxCox()`, `step_bs()`, `step_harmonic()`, `step_hyperbolic()`, `step_inverse()`, `step_invlogit()`, `step_logit()`, `step_log()`, `step_mutate()`, `step_ns()`, `step_percentile()`, `step_poly()`, `step_relu()`, `step_sqrt()`

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
)

yj_transform <- step_YeoJohnson(rec, all_numeric())

yj_estimates <- prep(yj_transform, training = biomass_tr)

yj_te <- bake(yj_estimates, biomass_te)

plot(density(biomass_te$sulfur), main = "before")
plot(density(yj_te$sulfur), main = "after")

tidy(yj_transform, number = 1)
tidy(yj_estimates, number = 1)
```

`step_zv` *Zero Variance Filter*

Description

`step_zv` creates a *specification* of a recipe step that will remove variables that contain only a single value.

Usage

```
step_zv(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  group = NULL,
  removals = NULL,
  skip = FALSE,
  id = rand_id("zv")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`group`	An optional character string or call to `dplyr::vars()` that can be used to specify a group(s) within which to identify variables that contain only a single value. If the grouping variables are contained in terms selector, they will not be considered for removal.
`removals`	A character string that contains the names of columns that should be removed. These values are not determined until `prep()` is called.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This step can potentially remove columns from the data set. This may cause issues for subsequent steps in your recipe if the missing columns are specifically referenced by name. To avoid this, see the advice in the *Tips for saving recipes and filtering columns* section of selections.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (the columns that will be removed) is returned.

Case weights

The underlying operation does not allow for case weights.

See Also

Other variable filter steps: `step_corr()`, `step_filter_missing()`, `step_lincomb()`, `step_nzv()`, `step_rm()`, `step_select()`

Examples

```
data(biomass, package = "modeldata")

biomass$one_value <- 1

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

rec <- recipe(HHV ~ carbon + hydrogen + oxygen +
  nitrogen + sulfur + one_value,
data = biomass_tr
)

zv_filter <- rec %>%
  step_zv(all_predictors())

filter_obj <- prep(zv_filter, training = biomass_tr)

filtered_te <- bake(filter_obj, biomass_te)
any(names(filtered_te) == "one_value")

tidy(zv_filter, number = 1)
tidy(filter_obj, number = 1)
```

`summary.recipe` *Summarize a recipe*

Description

This function prints the current set of variables/features and some of their characteristics.

Usage

```
## S3 method for class 'recipe'
summary(object, original = FALSE, ...)
```

Arguments

`object`	A `recipe` object
`original`	A logical: show the current set of variables or the original set when the recipe was defined.
`...`	further arguments passed to or from other methods (not currently used).

Details

Note that, until the recipe has been trained, the current and original variables are the same.

It is possible for variables to have multiple roles by adding them with `add_role()`. If a variable has multiple roles, it will have more than one row in the summary tibble.

Value

A tibble with columns `variable`, `type`, `role`, and `source`. When `original = TRUE`, an additional column is included named `required_to_bake` (based on the results of `update_role_requirements()`).

See Also

`recipe()` `prep()`

Examples

```
rec <- recipe(~., data = USArrests)
summary(rec)
rec <- step_pca(rec, all_numeric(), num_comp = 3)
summary(rec) # still the same since not yet trained
rec <- prep(rec, training = USArrests)
summary(rec)
```

`tidy.step_BoxCox` *Tidy the Result of a Recipe*

Description

`tidy` will return a data frame that contains information regarding a recipe or operation within the recipe (when a `tidy` method for the operation exists).

Usage

```
## S3 method for class 'step_BoxCox'
tidy(x, ...)

## S3 method for class 'step_YeoJohnson'
tidy(x, ...)

## S3 method for class 'step_arrange'
tidy(x, ...)

## S3 method for class 'step_bin2factor'
tidy(x, ...)

## S3 method for class 'step_bs'
tidy(x, ...)

## S3 method for class 'step_center'
tidy(x, ...)

## S3 method for class 'check_class'
tidy(x, ...)

## S3 method for class 'step_classdist'
tidy(x, ...)

## S3 method for class 'check_cols'
tidy(x, ...)

## S3 method for class 'step_corr'
tidy(x, ...)

## S3 method for class 'step_count'
tidy(x, ...)

## S3 method for class 'step_cut'
tidy(x, ...)

## S3 method for class 'step_date'
```

```
tidy(x, ...)

## S3 method for class 'step_depth'
tidy(x, ...)

## S3 method for class 'step_discretize'
tidy(x, ...)

## S3 method for class 'step_dummy'
tidy(x, ...)

## S3 method for class 'step_dummy_multi_choice'
tidy(x, ...)

## S3 method for class 'step_dummy_extract'
tidy(x, ...)

## S3 method for class 'step_factor2string'
tidy(x, ...)

## S3 method for class 'step_filter'
tidy(x, ...)

## S3 method for class 'step_filter_missing'
tidy(x, ...)

## S3 method for class 'step_geodist'
tidy(x, ...)

## S3 method for class 'step_harmonic'
tidy(x, ...)

## S3 method for class 'step_holiday'
tidy(x, ...)

## S3 method for class 'step_hyperbolic'
tidy(x, ...)

## S3 method for class 'step_ica'
tidy(x, ...)

## S3 method for class 'step_impute_bag'
tidy(x, ...)

## S3 method for class 'step_impute_knn'
tidy(x, ...)

## S3 method for class 'step_impute_linear'
```

```
tidy(x, ...)

## S3 method for class 'step_impute_lower'
tidy(x, ...)

## S3 method for class 'step_impute_mean'
tidy(x, ...)

## S3 method for class 'step_impute_median'
tidy(x, ...)

## S3 method for class 'step_impute_mode'
tidy(x, ...)

## S3 method for class 'step_impute_roll'
tidy(x, ...)

## S3 method for class 'step_integer'
tidy(x, ...)

## S3 method for class 'step_interact'
tidy(x, ...)

## S3 method for class 'step_inverse'
tidy(x, ...)

## S3 method for class 'step_invlogit'
tidy(x, ...)

## S3 method for class 'step_isomap'
tidy(x, ...)

## S3 method for class 'step_kpca'
tidy(x, ...)

## S3 method for class 'step_kpca_poly'
tidy(x, ...)

## S3 method for class 'step_kpca_rbf'
tidy(x, ...)

## S3 method for class 'step_lincomb'
tidy(x, ...)

## S3 method for class 'step_log'
tidy(x, ...)

## S3 method for class 'step_logit'
```

```
tidy(x, ...)

## S3 method for class 'check_missing'
tidy(x, ...)

## S3 method for class 'step_mutate'
tidy(x, ...)

## S3 method for class 'step_mutate_at'
tidy(x, ...)

## S3 method for class 'step_indicate_na'
tidy(x, ...)

## S3 method for class 'step_naomit'
tidy(x, ...)

## S3 method for class 'check_new_values'
tidy(x, ...)

## S3 method for class 'step_nnmf'
tidy(x, ...)

## S3 method for class 'step_nnmf_sparse'
tidy(x, ...)

## S3 method for class 'step_normalize'
tidy(x, ...)

## S3 method for class 'step_novel'
tidy(x, ...)

## S3 method for class 'step_ns'
tidy(x, ...)

## S3 method for class 'step_num2factor'
tidy(x, ...)

## S3 method for class 'step_nzv'
tidy(x, ...)

## S3 method for class 'step_ordinalscore'
tidy(x, ...)

## S3 method for class 'step_other'
tidy(x, ...)

## S3 method for class 'step_pca'
```

```
tidy(x, type = "coef", ...)

## S3 method for class 'step_percentile'
tidy(x, ...)

## S3 method for class 'step_pls'
tidy(x, ...)

## S3 method for class 'step_poly'
tidy(x, ...)

## S3 method for class 'step_profile'
tidy(x, ...)

## S3 method for class 'step_range'
tidy(x, ...)

## S3 method for class 'check_range'
tidy(x, ...)

## S3 method for class 'step_ratio'
tidy(x, ...)

## S3 method for class 'step_regex'
tidy(x, ...)

## S3 method for class 'step_relevel'
tidy(x, ...)

## S3 method for class 'step_relu'
tidy(x, ...)

## S3 method for class 'step_rename'
tidy(x, ...)

## S3 method for class 'step_rename_at'
tidy(x, ...)

## S3 method for class 'step_rm'
tidy(x, ...)

## S3 method for class 'step_sample'
tidy(x, ...)

## S3 method for class 'step_scale'
tidy(x, ...)

## S3 method for class 'step_select'
```

```
tidy(x, ...)

## S3 method for class 'step_shuffle'
tidy(x, ...)

## S3 method for class 'step_slice'
tidy(x, ...)

## S3 method for class 'step_spatialsign'
tidy(x, ...)

## S3 method for class 'step_sqrt'
tidy(x, ...)

## S3 method for class 'step_string2factor'
tidy(x, ...)

## S3 method for class 'recipe'
tidy(x, number = NA, id = NA, ...)

## S3 method for class 'step'
tidy(x, ...)

## S3 method for class 'check'
tidy(x, ...)

## S3 method for class 'step_time'
tidy(x, ...)

## S3 method for class 'step_unknown'
tidy(x, ...)

## S3 method for class 'step_unorder'
tidy(x, ...)

## S3 method for class 'step_window'
tidy(x, ...)

## S3 method for class 'step_zv'
tidy(x, ...)
```

Arguments

`x`	A `recipe` object, step, or check (trained or otherwise).
`...`	Not currently used.
`type`	For `step_pca`, either `"coef"` (for the variable loadings per component) or `"variance"` (how much variance does each component account for).
`number`	An integer or `NA`. If missing and `id` is not provided, the return value is a list of

	the operations in the recipe. If a number is given, a `tidy` method is executed for that operation in the recipe (if it exists). `number` must not be provided if `id` is.
`id`	A character string or `NA`. If missing and `number` is not provided, the return value is a list of the operations in the recipe. If a character string is given, a `tidy` method is executed for that operation in the recipe (if it exists). `id` must not be provided if `number` is.

Value

A tibble with columns that vary depending on what `tidy` method is executed. When `number` and `id` are `NA`, a tibble with columns `number` (the operation iteration), `operation` (either "step" or "check"), `type` (the method, e.g. "nzv", "center"), a logical column called `trained` for whether the operation has been estimated using `prep`, a logical for `skip`, and a character column `id`.

Examples

```
data(Sacramento, package = "modeldata")

Sacramento_rec <- recipe(~., data = Sacramento) %>%
  step_other(all_nominal(), threshold = 0.05, other = "another") %>%
  step_center(all_numeric()) %>%
  step_dummy(all_nominal()) %>%
  check_cols(ends_with("ude"), sqft, price)

tidy(Sacramento_rec)

tidy(Sacramento_rec, number = 2)
tidy(Sacramento_rec, number = 3)

Sacramento_rec_trained <- prep(Sacramento_rec, training = Sacramento)

tidy(Sacramento_rec_trained)
tidy(Sacramento_rec_trained, number = 3)
tidy(Sacramento_rec_trained, number = 4)
```

`update.step` *Update a recipe step*

Description

This `step` method for `update()` takes named arguments as `...` who's values will replace the elements of the same name in the actual step.

Usage

```
## S3 method for class 'step'
update(object, ...)
```

Arguments

`object`	A recipe `step`.
`...`	Key-value pairs where the keys match up with names of elements in the step, and the values are the new values to update the step with.

Details

For a step to be updated, it must not already have been trained. Otherwise, conflicting information can arise between the data returned from `bake(object, new_data = NULL)` and the information in the step.

Examples

```
data(biomass, package = "modeldata")

biomass_tr <- biomass[biomass$dataset == "Training", ]
biomass_te <- biomass[biomass$dataset == "Testing", ]

# Create a recipe using step_bs() with degree = 3
rec <- recipe(
  HHV ~ carbon + hydrogen + oxygen + nitrogen + sulfur,
  data = biomass_tr
) %>%
  step_bs(carbon, hydrogen, degree = 3)

# Update the step to use degree = 4
rec2 <- rec
rec2$steps[[1]] <- update(rec2$steps[[1]], degree = 4)

# Prep both recipes
rec_prepped <- prep(rec, training = biomass_tr)
rec2_prepped <- prep(rec2, training = biomass_tr)

# Juice both to see what changed
bake(rec_prepped, new_data = NULL)
bake(rec2_prepped, new_data = NULL)

# Cannot update a recipe step that has been trained!
## Not run:
update(rec_prepped$steps[[1]], degree = 4)

## End(Not run)
```

`update_role_requirements`

Update role specific requirements

Description

`update_role_requirements()` allows you to fine tune requirements of the various roles you might come across in recipes (see `update_role()` for general information about roles). Role requirements can only be altered for roles that exist in the *original* data supplied to `recipe()`, they are not applied to columns computed by steps.

Like `update_role()`, `update_role_requirements()` is applied to the recipe *immediately*, unlike the `step_*()` functions which do most of their work at `prep()` time.

Usage

```
update_role_requirements(recipe, role, ..., bake = NULL)
```

Arguments

`recipe`	A recipe.
`role`	A string representing the role that you'd like to modify the requirements of. This must be a role that already exists in the recipe.
`...`	These dots are for future extensions and must be empty.
`bake`	At `bake()` time, should a check be done to ensure that all columns of this role that were supplied to `recipe()` also be present in the `new_data` supplied to `bake()`? Must be a single `TRUE` or `FALSE`. The default, `NULL`, won't modify this requirement. The following represents the default bake time requirements of specific types of roles: • `"outcome"`: Not required at bake time. Can't be changed. • `"predictor"`: Required at bake time. Can't be changed. • `"case_weights"`: Not required at bake time by default. • `NA`: Required at bake time by default. • Custom roles: Required at bake time by default.

Examples

```
df <- tibble(y = c(1, 2, 3), x = c(4, 5, 6), var = c("a", "b", "c"))

# Let's assume that you have a `var` column that isn't used in the recipe.
# We typically recommend that you remove this column before passing the
# `data` to `recipe()`, but for now let's pass it through and assign it an
# `"id"` role.
rec <- recipe(y ~ ., df) %>%
  update_role(var, new_role = "id") %>%
  step_center(x)

prepped <- prep(rec, df)

# Now assume you have some "new data" and you are ready to `bake()` it
# to prepare it for prediction purposes. Here, you might not have `var`
```

```
# available as a column because it isn't important to your model.
new_data <- df[c("y", "x")]

# By default `var` is required at `bake()` time because we don't know if
# you actually use it in the recipe or not
try(bake(prepped, new_data))

# You can turn off this check by using `update_role_requirements()` and
# setting `bake = FALSE` for the `"id"` role. We recommend doing this on
# the original unprepped recipe, but it will also work on a prepped recipe.
rec <- update_role_requirements(rec, "id", bake = FALSE)
prepped <- prep(rec, df)

# Now you can `bake()` on `new_data` even though `var` is missing
bake(prepped, new_data)
```

Index