Package: parsnip (via r-universe)

October 4, 2024

Title A Common API to Modeling and Analysis Functions

Version 1.2.1.9002

Maintainer Max Kuhn <max@posit.co>

Description A common interface is provided to allow users to specify a model without having to remember the different argument names across different functions or computational engines (e.g. 'R', 'Spark', 'Stan', 'H2O', etc).

License MIT + file LICENSE

URL https://github.com/tidymodels/parsnip, https://parsnip.tidymodels.org/

BugReports https://github.com/tidymodels/parsnip/issues

Depends R (>= 3.6)

Imports cli, dplyr (>= 1.1.0), generics (>= 0.1.2), ggplot2, globals, glue, hardhat (>= 1.4.0), lifecycle, magrittr, pillar, prettyunits, purrr (>= 1.0.0), rlang (>= 1.1.0), sparsevctrs (>= 0.1.0.9002), stats, tibble (>= 2.1.1), tidyr (>= 1.3.0), utils, vctrs (>= 0.6.0), withr

Suggests C50, bench, covr, dials (>= 1.1.0), earth, ggrepel, keras, kernlab, kknn, knitr, LiblineaR, MASS, Matrix, methods, mgcv, modeldata, nlme, prodlim, ranger (>= 0.12.0), remotes, rmarkdown, rpart, sparklyr (>= 1.0.0), survival, tensorflow, testthat (>= 3.0.0), xgboost (>= 1.5.0.1)

VignetteBuilder knitr

ByteCompile true

Config/Needs/website C50, dbarts, earth, glmnet, keras, kernlab, kknn, LiblineaR, mgcv, nnet, parsnip, randomForest, ranger, rpart, rstanarm, tidymodels/tidymodels, tidyverse/tidytemplate, rstudio/reticulate, xgboost

Config/rcmdcheck/ignore-inconsequential-notes true

Config/testthat/edition 3

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

Remotes r-lib/sparsevctrs

RoxygenNote 7.3.2

Repository https://tidymodels.r-universe.dev

RemoteUrl https://github.com/tidymodels/parsnip

RemoteRef HEAD

RemoteSha 5ce414eba2156bdf4a71ce4a72f379be6e8c9c85

Contents

`.extract_surv_status` *Extract survival status*

Description

Extract the status from a `survival::Surv()` object.

Arguments

`surv`	A single `survival::Surv()` object.

Value

A numeric vector.

`.extract_surv_time` *Extract survival time*

Description

Extract the time component(s) from a `survival::Surv()` object.

Arguments

`surv`	A single `survival::Surv()` object.

Value

A vector when the type is `"right"` or `"left"` and a tibble otherwise.

`.model_param_name_key` *Translate names of model tuning parameters*

Description

This function creates a key that connects the identifiers users make for tuning parameter names, the standardized parsnip parameter names, and the argument names to the underlying fit function for the engine.

Usage

```
.model_param_name_key(object, as_tibble = TRUE)
```

Arguments

`object`	A workflow or parsnip model specification.
`as_tibble`	A logical. Should the results be in a tibble (the default) or in a list that can facilitate renaming grid objects?

Value

A tibble with columns `user`, `parsnip`, and `engine`, or a list with named character vectors `user_to_parsnip` and `parsnip_to_engine`.

Examples

```
mod <-
 linear_reg(penalty = tune("regularization"), mixture = tune()) %>%
 set_engine("glmnet")

mod %>% .model_param_name_key()

rn <- mod %>% .model_param_name_key(as_tibble = FALSE)
rn

grid <- tidyr::crossing(regularization = c(0, 1), mixture = (0:3) / 3)

grid %>%
  dplyr::rename(!!!rn$user_to_parsnip)

grid %>%
  dplyr::rename(!!!rn$user_to_parsnip) %>%
  dplyr::rename(!!!rn$parsnip_to_engine)
```

`add_rowindex` *Add a column of row numbers to a data frame*

Description

Add a column of row numbers to a data frame

Usage

```
add_rowindex(x)
```

Arguments

`x`	A data frame

Value

The same data frame with a column of 1-based integers named `.row`.

Examples

```
mtcars %>% add_rowindex()
```

augment.model_fit *Augment data with predictions*

Description

`augment()` will add column(s) for predictions to the given data.

Usage

```
## S3 method for class 'model_fit'
augment(x, new_data, eval_time = NULL, ...)
```

Arguments

`x`	A model fit produced by `fit.model_spec()` or `fit_xy.model_spec()`.
`new_data`	A data frame or matrix.
`eval_time`	For censored regression models, a vector of time points at which the survival probability is estimated.
`...`	Not currently used.

Details

Regression:

For regression models, a `.pred` column is added. If `x` was created using `fit.model_spec()` and `new_data` contains a regression outcome column, a `.resid` column is also added.

Classification:

For classification models, the results can include a column called `.pred_class` as well as class probability columns named `.pred_{level}`. This depends on what type of prediction types are available for the model.

Censored Regression:

For these models, predictions for the expected time and survival probability are created (if the model engine supports them). If the model supports survival prediction, the `eval_time` argument is required.

If survival predictions are created and `new_data` contains a `survival::Surv()` object, additional columns are added for inverse probability of censoring weights (IPCW) are also created (see `tidymodels.org` page in the references below). This enables the user to compute performance metrics in the **yardstick** package.

References

https://www.tidymodels.org/learn/statistics/survival-metrics/

Examples

```
car_trn <- mtcars[11:32,]
car_tst <- mtcars[ 1:10,]

reg_form <-
  linear_reg() %>%
  set_engine("lm") %>%
  fit(mpg ~ ., data = car_trn)
reg_xy <-
  linear_reg() %>%
  set_engine("lm") %>%
  fit_xy(car_trn[, -1], car_trn$mpg)

augment(reg_form, car_tst)
augment(reg_form, car_tst[, -1])

augment(reg_xy, car_tst)
augment(reg_xy, car_tst[, -1])

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

data(two_class_dat, package = "modeldata")
cls_trn <- two_class_dat[-(1:10), ]
cls_tst <- two_class_dat[  1:10 , ]

cls_form <-
  logistic_reg() %>%
  set_engine("glm") %>%
  fit(Class ~ ., data = cls_trn)
cls_xy <-
  logistic_reg() %>%
  set_engine("glm") %>%
  fit_xy(cls_trn[, -3],
  cls_trn$Class)

augment(cls_form, cls_tst)
augment(cls_form, cls_tst[, -3])

augment(cls_xy, cls_tst)
augment(cls_xy, cls_tst[, -3])
```

`autoplot.model_fit` *Create a ggplot for a model object*

Description

This method provides a good visualization method for model results. Currently, only methods for glmnet models are implemented.

Usage

```
## S3 method for class 'model_fit'
autoplot(object, ...)

## S3 method for class 'glmnet'
autoplot(object, ..., min_penalty = 0, best_penalty = NULL, top_n = 3L)
```

Arguments

`object`	A model fit object.
`...`	For `autoplot.glmnet()`, options to pass to `ggrepel::geom_label_repel()`. Otherwise, this argument is ignored.
`min_penalty`	A single, non-negative number for the smallest penalty value that should be shown in the plot. If left `NULL`, the whole data range is used.
`best_penalty`	A single, non-negative number that will show a vertical line marker. If left `NULL`, no line is shown. When this argument is used, the **ggrepl** package is required.
`top_n`	A non-negative integer for how many model predictors to label. The top predictors are ranked by their absolute coefficient value. For multinomial or multivariate models, the `top_n` terms are selected within class or response, respectively.

Details

The **glmnet** package will need to be attached or loaded for its `autoplot()` method to work correctly.

Value

A ggplot object with penalty on the x-axis and coefficients on the y-axis. For multinomial or multivariate models, the plot is faceted.

`auto_ml` *Automatic Machine Learning*

Description

`auto_ml()` defines an automated searching and tuning process where many models of different families are trained and ranked given their performance on the training data.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `h2o`12

1 The default engine. 2 Requires a parsnip extension package for classification and regression.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
auto_ml(mode = "unknown", engine = "h2o")
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`engine`	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
auto_ml(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `h2o engine details`

`bag_mars` *Ensembles of MARS models*

Description

`bag_mars()` defines an ensemble of generalized linear models that use artificial features for some predictors. These features resemble hinge functions and the result is a model that is a segmented regression in small dimensions. This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `earth`12

1 The default engine. 2 Requires a parsnip extension package for classification and regression.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
bag_mars(
  mode = "unknown",
  num_terms = NULL,
  prod_degree = NULL,
  prune_method = NULL,
  engine = "earth"
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`num_terms`	The number of features that will be retained in the final model, including the intercept.
`prod_degree`	The highest possible interaction degree.
`prune_method`	The pruning method.
`engine`	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
bag_mars(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `earth engine details`

bag_mlp *Ensembles of neural networks*

Description

bag_mlp() defines an ensemble of single layer, feed-forward neural networks. This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- nnet12

1 The default engine. 2 Requires a parsnip extension package for classification and regression.

More information on how **parsnip** is used for modeling is at https://www.tidymodels.org/.

Usage

```
bag_mlp(
  mode = "unknown",
  hidden_units = NULL,
  penalty = NULL,
  epochs = NULL,
  engine = "nnet"
)
```

Arguments

mode	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
hidden_units	An integer for the number of units in the hidden model.
penalty	A non-negative numeric value for the amount of weight decay.
epochs	An integer for the number of training iterations.
engine	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See set_engine() for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the fit() function is used with the data.

Each of the arguments in this function other than mode and engine are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
bag_mlp(argument = !!value)
```

References

https://www.tidymodels.org, *Tidy Modeling with R*, searchable table of parsnip models

See Also

fit(), set_engine(), update(), nnet engine details

---

bag_tree *Ensembles of decision trees*

Description

bag_tree() defines an ensemble of decision trees. This function can fit classification, regression, and censored regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- rpart12
- C5.02

1 The default engine. 2 Requires a parsnip extension package for censored regression, classification, and regression.

More information on how **parsnip** is used for modeling is at https://www.tidymodels.org/.

Usage

```
bag_tree(
  mode = "unknown",
  cost_complexity = 0,
  tree_depth = NULL,
  min_n = 2,
  class_cost = NULL,
  engine = "rpart"
)
```

Arguments

mode	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", "classification", or "censored regression".
cost_complexity	A positive number for the the cost/complexity parameter (a.k.a. Cp) used by CART models (specific engines only).
tree_depth	An integer for the maximum depth of the tree (i.e. number of splits) (specific engines only).

`min_n`	An integer for the minimum number of data points in a node that is required for the node to be split further.
`class_cost`	A non-negative scalar for a class cost (where a cost of 1 means no extra cost). This is useful for when the first level of the outcome factor is the minority class. If this is not the case, values between zero and one can be used to bias to the second level of the factor.
`engine`	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
bag_tree(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `rpart engine details`, `C5.0 engine details`

`bart` *Bayesian additive regression trees (BART)*

Description

`bart()` defines a tree ensemble model that uses Bayesian analysis to assemble the ensemble. This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `dbarts`1

1 The default engine.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
bart(
  mode = "unknown",
  engine = "dbarts",
  trees = NULL,
  prior_terminal_node_coef = NULL,
  prior_terminal_node_expo = NULL,
  prior_outcome_range = NULL
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`engine`	A single character string specifying what computational engine to use for fitting.
`trees`	An integer for the number of trees contained in the ensemble.
`prior_terminal_node_coef`	A coefficient for the prior probability that a node is a terminal node. Values are usually between 0 and one with a default of 0.95. This affects the baseline probability; smaller numbers make the probabilities larger overall. See Details below.
`prior_terminal_node_expo`	An exponent in the prior probability that a node is a terminal node. Values are usually non-negative with a default of 2 This affects the rate that the prior probability decreases as the depth of the tree increases. Larger values make deeper trees less likely.
`prior_outcome_range`	A positive value that defines the width of a prior that the predicted outcome is within a certain range. For regression it is related to the observed range of the data; the prior is the number of standard deviations of a Gaussian distribution defined by the observed range of the data. For classification, it is defined as the range of +/-3 (assumed to be on the logit scale). The default value is 2.

Details

The prior for the terminal node probability is expressed as `prior = a * (1 + d)^(-b)` where `d` is the depth of the node, `a` is `prior_terminal_node_coef` and `b` is `prior_terminal_node_expo`. See the Examples section below for an example graph of the prior probability of a terminal node for different values of these parameters.

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
bart(argument = !!value)
```

References

https://www.tidymodels.org, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `dbarts engine details`

Examples

```
show_engines("bart")

bart(mode = "regression", trees = 5)

# ------------------------------------------------------------------------------
# Examples for terminal node prior

library(ggplot2)
library(dplyr)

prior_test <- function(coef = 0.95, expo = 2, depths = 1:10) {
  tidyr::crossing(coef = coef, expo = expo, depth = depths) %>%
    mutate(
      `terminial node prior` = coef * (1 + depth)^(-expo),
      coef = format(coef),
      expo = format(expo))
}

prior_test(coef = c(0.05, 0.5, .95), expo = c(1/2, 1, 2)) %>%
  ggplot(aes(depth, `terminial node prior`, col = coef)) +
  geom_line() +
  geom_point() +
  facet_wrap(~ expo)
```

`boost_tree` *Boosted trees*

Description

`boost_tree()` defines a model that creates a series of decision trees forming an ensemble. Each tree depends on the results of previous trees. All trees in the ensemble are combined to produce a final prediction. This function can fit classification, regression, and censored regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `xgboost`1
- `C5.0`
- `h2o`2

- lightgbm2
- mboost2
- spark

1 The default engine. 2 Requires a parsnip extension package for censored regression, classification, and regression.

More information on how **parsnip** is used for modeling is at https://www.tidymodels.org/.

Usage

```
boost_tree(
  mode = "unknown",
  engine = "xgboost",
  mtry = NULL,
  trees = NULL,
  min_n = NULL,
  tree_depth = NULL,
  learn_rate = NULL,
  loss_reduction = NULL,
  sample_size = NULL,
  stop_iter = NULL
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", "classification", or "censored regression".
`engine`	A single character string specifying what computational engine to use for fitting.
`mtry`	A number for the number (or proportion) of predictors that will be randomly sampled at each split when creating the tree models (specific engines only).
`trees`	An integer for the number of trees contained in the ensemble.
`min_n`	An integer for the minimum number of data points in a node that is required for the node to be split further.
`tree_depth`	An integer for the maximum depth of the tree (i.e. number of splits) (specific engines only).
`learn_rate`	A number for the rate at which the boosting algorithm adapts from iteration-to-iteration (specific engines only). This is sometimes referred to as the shrinkage parameter.
`loss_reduction`	A number for the reduction in the loss function required to split further (specific engines only).
`sample_size`	A number for the number (or proportion) of data that is exposed to the fitting routine. For `xgboost`, the sampling is done at each iteration while `C5.0` samples once during training.
`stop_iter`	The number of iterations without improvement before stopping (specific engines only).

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
boost_tree(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `xgboost engine details`, `C5.0 engine details`, `h2o engine details`, `lightgbm engine details`, `mboost engine details`, `spark engine details`, `xgb_train()`, `C5.0_train()`

Examples

```
show_engines("boost_tree")

boost_tree(mode = "classification", trees = 20)
```

C5_rules — *C5.0 rule-based classification models*

Description

`C5_rules()` defines a model that derives feature rules from a tree for prediction. A single tree or boosted ensemble can be used. This function can fit classification models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `C5.0`12

1 The default engine. 2 Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
C5_rules(mode = "classification", trees = NULL, min_n = NULL, engine = "C5.0")
```

Arguments

mode	A single character string for the type of model. The only possible value for this model is "classification".
trees	A non-negative integer (no greater than 100) for the number of members of the ensemble.
min_n	An integer greater between zero and nine for the minimum number of data points in a node that are required for the node to be split further.
engine	A single character string specifying what computational engine to use for fitting.

Details

C5.0 is a classification model that is an extension of the C4.5 model of Quinlan (1993). It has tree- and rule-based versions that also include boosting capabilities. `C5_rules()` enables the version of the model that uses a series of rules (see the examples below). To make a set of rules, an initial C5.0 tree is created and flattened into rules. The rules are pruned, simplified, and ordered. Rule sets are created within each iteration of boosting.

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
C5_rules(argument = !!value)
```

References

Quinlan R (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers.

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`C50::C5.0()`, `C50::C5.0Control()`, `fit()`, `set_engine()`, `update()`, `C5.0 engine details`

Examples

```
show_engines("C5_rules")

C5_rules()
```

`case_weights` *Using case weights with parsnip*

Description

Case weights are positive numeric values that influence how much each data point has during the model fitting process. 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 parsnip, the `fit()` and `fit_xy()` functions contain a `case_weight` argument that takes these data. For Spark models, the argument value should be a character value.

See Also

`frequency_weights()`, `importance_weights()`, `fit()`, `fit_xy()`

`case_weights_allowed` *Determine if case weights are used*

Description

Not all modeling engines can incorporate case weights into their calculations. This function can determine whether they can be used.

Usage

```
case_weights_allowed(spec)
```

Arguments

`spec`	A parsnip model specification.

Value

A single logical.

Examples

```
case_weights_allowed(linear_reg())
case_weights_allowed(linear_reg(engine = "keras"))
```

`control_parsnip` *Control the fit function*

Description

Pass options to the `fit.model_spec()` function to control its output and computations

Usage

```
control_parsnip(verbosity = 1L, catch = FALSE)
```

Arguments

`verbosity`	An integer to control how verbose the output is. For a value of zero, no messages or output are shown when packages are loaded or when the model is fit. For a value of 1, package loading is quiet but model fits can produce output to the screen (depending on if they contain their own `verbose`-type argument). For a value of 2 or more, any output at all is displayed and the execution time of the fit is recorded and printed.
`catch`	A logical where a value of `TRUE` will evaluate the model inside of `try(, silent = TRUE)`. If the model fails, an object is still returned (without an error) that inherits the class "try-error".

Value

An S3 object with class "control_parsnip" that is a named list with the results of the function call

Examples

```
control_parsnip(verbosity = 2L)
```

contr_one_hot *Contrast function for one-hot encodings*

Description

This contrast function produces a model matrix with indicator columns for each level of each factor.

Usage

```
contr_one_hot(n, contrasts = TRUE, sparse = FALSE)
```

Arguments

n	A vector of character factor levels or the number of unique levels.
contrasts	This argument is for backwards compatibility and only the default of TRUE is supported.
sparse	This argument is for backwards compatibility and only the default of FALSE is supported.

Details

By default, `model.matrix()` generates binary indicator variables for factor predictors. When the formula does not remove an intercept, an incomplete set of indicators are created; no indicator is made for the first level of the factor.

For example, `species` and `island` both have three levels but `model.matrix()` creates two indicator variables for each:

```
library(dplyr)
library(modeldata)
data(penguins)

levels(penguins$species)

## [1] "Adelie"    "Chinstrap" "Gentoo"

levels(penguins$island)

## [1] "Biscoe"    "Dream"     "Torgersen"

model.matrix(~ species + island, data = penguins) %>%
  colnames()

## [1] "(Intercept)"      "speciesChinstrap" "speciesGentoo"    "islandDream"
## [5] "islandTorgersen"
```

For a formula with no intercept, the first factor is expanded to indicators for *all* factor levels but all other factors are expanded to all but one (as above):

```
model.matrix(~ 0 + species + island, data = penguins) %>%
  colnames()
```

```
## [1] "speciesAdelie"    "speciesChinstrap" "speciesGentoo"    "islandDream"
## [5] "islandTorgersen"
```

For inference, this hybrid encoding can be problematic.

To generate all indicators, use this contrast:

```
# Switch out the contrast method
old_contr <- options("contrasts")$contrasts
new_contr <- old_contr
new_contr["unordered"] <- "contr_one_hot"
options(contrasts = new_contr)

model.matrix(~ species + island, data = penguins) %>%
  colnames()
```

```
## [1] "(Intercept)"      "speciesAdelie"    "speciesChinstrap" "speciesGentoo"
## [5] "islandBiscoe"     "islandDream"      "islandTorgersen"
```

```
options(contrasts = old_contr)
```

Removing the intercept here does not affect the factor encodings.

Value

A diagonal matrix that is n-by-n.

`ctree_train` *A wrapper function for conditional inference tree models*

Description

These functions are slightly different APIs for `partykit::ctree()` and `partykit::cforest()` that have several important arguments as top-level arguments (as opposed to being specified in `partykit::ctree_control()`).

Usage

```
ctree_train(
  formula,
  data,
  weights = NULL,
  minsplit = 20L,
  maxdepth = Inf,
  teststat = "quadratic",
  testtype = "Bonferroni",
  mincriterion = 0.95,
  ...
)

cforest_train(
  formula,
  data,
  weights = NULL,
  minsplit = 20L,
  maxdepth = Inf,
  teststat = "quadratic",
  testtype = "Univariate",
  mincriterion = 0,
  mtry = ceiling(sqrt(ncol(data) - 1)),
  ntree = 500L,
  ...
)
```

Arguments

`formula`	A symbolic description of the model to be fit.
`data`	A data frame containing the variables in the model.
`weights`	A vector of weights whose length is the same as `nrow(data)`. For `partykit::ctree()` models, these are required to be non-negative integers while for `partykit::cforest()` they can be non-negative integers or doubles.
`minsplit`	The minimum sum of weights in a node in order to be considered for splitting.
`maxdepth`	maximum depth of the tree. The default `maxdepth = Inf` means that no restrictions are applied to tree sizes.
`teststat`	A character specifying the type of the test statistic to be applied.
`testtype`	A character specifying how to compute the distribution of the test statistic.
`mincriterion`	The value of the test statistic (for `testtype == "Teststatistic"`), or 1 - p-value (for other values of `testtype`) that must be exceeded in order to implement a split.
`...`	Other options to pass to `partykit::ctree()` or `partykit::cforest()`.
`mtry`	Number of input variables randomly sampled as candidates at each node for random forest like algorithms. The default `mtry = Inf` means that no random selection takes place.

`ntree`	Number of trees to grow in a forest.

Value

An object of class `party` (for `ctree`) or `cforest`.

Examples

```
if (rlang::is_installed(c("modeldata", "partykit"))) {
  data(bivariate, package = "modeldata")
  ctree_train(Class ~ ., data = bivariate_train)
  ctree_train(Class ~ ., data = bivariate_train, maxdepth = 1)
}
```

`cubist_rules` *Cubist rule-based regression models*

Description

`cubist_rules()` defines a model that derives simple feature rules from a tree ensemble and creates regression models within each rule. This function can fit regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `Cubist`12

1 The default engine. 2 Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at https://www.tidymodels.org/.

Usage

```
cubist_rules(
  mode = "regression",
  committees = NULL,
  neighbors = NULL,
  max_rules = NULL,
  engine = "Cubist"
)
```

Arguments

`mode`	A single character string for the type of model. The only possible value for this model is "regression".
`committees`	A non-negative integer (no greater than 100) for the number of members of the ensemble.
`neighbors`	An integer between zero and nine for the number of training set instances that are used to adjust the model-based prediction.

`max_rules`	The largest number of rules.
`engine`	A single character string specifying what computational engine to use for fitting.

Details

Cubist is a rule-based ensemble regression model. A basic model tree (Quinlan, 1992) is created that has a separate linear regression model corresponding for each terminal node. The paths along the model tree are flattened into rules and these rules are simplified and pruned. The parameter `min_n` is the primary method for controlling the size of each tree while `max_rules` controls the number of rules.

Cubist ensembles are created using *committees*, which are similar to boosting. After the first model in the committee is created, the second model uses a modified version of the outcome data based on whether the previous model under- or over-predicted the outcome. For iteration *m*, the new outcome `y*` is computed using

$$y^*_{(m)} = y - (\widehat{y}_{(m-1)} - y)$$

If a sample is under-predicted on the previous iteration, the outcome is adjusted so that the next time it is more likely to be over-predicted to compensate. This adjustment continues for each ensemble iteration. See Kuhn and Johnson (2013) for details.

After the model is created, there is also an option for a post-hoc adjustment that uses the training set (Quinlan, 1993). When a new sample is predicted by the model, it can be modified by its nearest neighbors in the original training set. For *K* neighbors, the model-based predicted value is adjusted by the neighbor using:

$$\frac{1}{K}\sum_{\ell=1}^{K} w_\ell \left[t_\ell + (\widehat{y} - \widehat{t}_\ell)\right]$$

where `t` is the training set prediction and `w` is a weight that is inverse to the distance to the neighbor.

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
cubist_rules(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

Quinlan R (1992). "Learning with Continuous Classes." Proceedings of the 5th Australian Joint Conference On Artificial Intelligence, pp. 343-348.

Quinlan R (1993)."Combining Instance-Based and Model-Based Learning." Proceedings of the Tenth International Conference on Machine Learning, pp. 236-243.

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

See Also

`Cubist::cubist()`, `Cubist::cubistControl()`, `fit()`, `set_engine()`, `update()`, `Cubist engine details`

`decision_tree` *Decision trees*

Description

`decision_tree()` defines a model as a set of `if/then` statements that creates a tree-based structure. This function can fit classification, regression, and censored regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `rpart`¹²
- `C5.0`
- `partykit`²
- `spark`

1 The default engine. 2 Requires a parsnip extension package for censored regression, classification, and regression.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
decision_tree(
  mode = "unknown",
  engine = "rpart",
  cost_complexity = NULL,
  tree_depth = NULL,
  min_n = NULL
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", "classification", or "censored regression".
`engine`	A single character string specifying what computational engine to use for fitting.
`cost_complexity`	A positive number for the the cost/complexity parameter (a.k.a. `Cp`) used by CART models (specific engines only).
`tree_depth`	An integer for maximum depth of the tree.
`min_n`	An integer for the minimum number of data points in a node that are required for the node to be split further.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
decision_tree(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `rpart engine details`, `C5.0 engine details`, `partykit engine details`, `spark engine details`

Examples

```
show_engines("decision_tree")

decision_tree(mode = "classification", tree_depth = 5)
```

`descriptors` *Data Set Characteristics Available when Fitting Models*

Description

When using the `fit()` functions there are some variables that will be available for use in arguments. For example, if the user would like to choose an argument value based on the current number of rows in a data set, the `.obs()` function can be used. See Details below.

Usage

```
.cols()

.preds()

.obs()

.lvls()

.facts()

.x()

.y()

.dat()
```

Details

Existing functions:

- `.obs()`: The current number of rows in the data set.
- `.preds()`: The number of columns in the data set that is associated with the predictors prior to dummy variable creation.
- `.cols()`: The number of predictor columns available after dummy variables are created (if any).
- `.facts()`: The number of factor predictors in the data set.
- `.lvls()`: If the outcome is a factor, this is a table with the counts for each level (and `NA` otherwise).
- `.x()`: The predictors returned in the format given. Either a data frame or a matrix.
- `.y()`: The known outcomes returned in the format given. Either a vector, matrix, or data frame.
- `.dat()`: A data frame containing all of the predictors and the outcomes. If `fit_xy()` was used, the outcomes are attached as the column, `..y`.

For example, if you use the model formula `circumference ~ .` with the built-in `Orange` data, the values would be

```
.preds() =   2          (the 2 remaining columns in `Orange`)
.cols()  =   5          (1 numeric column + 4 from Tree dummy variables)
.obs()   = 35
.lvls()  =  NA          (no factor outcome)
.facts() =   1          (the Tree predictor)
.y()     = <vector>     (circumference as a vector)
.x()     = <data.frame> (The other 2 columns as a data frame)
.dat()   = <data.frame> (The full data set)
```

If the formula `Tree ~ .` were used:

```
.preds() =   2          (the 2 numeric columns in `Orange`)
.cols()  =   2          (same)
.obs()   = 35
.lvls()  =  c("1" = 7, "2" = 7, "3" = 7, "4" = 7, "5" = 7)
.facts() =   0
.y()     = <vector>     (Tree as a vector)
.x()     = <data.frame> (The other 2 columns as a data frame)
.dat()   = <data.frame> (The full data set)
```

To use these in a model fit, pass them to a model specification. The evaluation is delayed until the time when the model is run via `fit()` (and the variables listed above are available). For example:

```
library(modeldata)
data("lending_club")

rand_forest(mode = "classification", mtry = .cols() - 2)
```

When no descriptors are found, the computation of the descriptor values is not executed.

`discrim_flexible` *Flexible discriminant analysis*

Description

`discrim_flexible()` defines a model that fits a discriminant analysis model that can use nonlinear features created using multivariate adaptive regression splines (MARS). This function can fit classification models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `earth`12

1 The default engine. 2 Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
discrim_flexible(
  mode = "classification",
  num_terms = NULL,
  prod_degree = NULL,
  prune_method = NULL,
  engine = "earth"
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`num_terms`	The number of features that will be retained in the final model, including the intercept.
`prod_degree`	The highest possible interaction degree.
`prune_method`	The pruning method.
`engine`	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
discrim_flexible(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `earth engine details`

discrim_linear *Linear discriminant analysis*

Description

discrim_linear() defines a model that estimates a multivariate distribution for the predictors separately for the data in each class (usually Gaussian with a common covariance matrix). Bayes' theorem is used to compute the probability of each class, given the predictor values. This function can fit classification models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- MASS12
- mda2
- sda2
- sparsediscrim2

1 The default engine. 2 Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at https://www.tidymodels.org/.

Usage

```
discrim_linear(
  mode = "classification",
  penalty = NULL,
  regularization_method = NULL,
  engine = "MASS"
)
```

Arguments

mode	A single character string for the type of model. The only possible value for this model is "classification".
penalty	An non-negative number representing the amount of regularization used by some of the engines.
regularization_method	A character string for the type of regularized estimation. Possible values are: "diagonal", "min_distance", "shrink_cov", and "shrink_mean" (sparsediscrim engine only).
engine	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See set_engine() for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the fit() function is used with the data.

Each of the arguments in this function other than mode and engine are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
discrim_linear(argument = !!value)
```

References

https://www.tidymodels.org, *Tidy Modeling with R*, searchable table of parsnip models

See Also

fit(), set_engine(), update(), MASS engine details, mda engine details, sda engine details, sparsediscrim engine details

---

discrim_quad *Quadratic discriminant analysis*

Description

discrim_quad() defines a model that estimates a multivariate distribution for the predictors separately for the data in each class (usually Gaussian with separate covariance matrices). Bayes' theorem is used to compute the probability of each class, given the predictor values. This function can fit classification models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- MASS12
- sparsediscrim2

1 The default engine. 2 Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at https://www.tidymodels.org/.

Usage

```
discrim_quad(
  mode = "classification",
  regularization_method = NULL,
  engine = "MASS"
)
```

Arguments

`mode`	A single character string for the type of model. The only possible value for this model is "classification".
`regularization_method`	A character string for the type of regularized estimation. Possible values are: `"diagonal"`, `"shrink_cov"`, and `"shrink_mean"` (`sparsediscrim` engine only).
`engine`	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
discrim_quad(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `MASS engine details`, `sparsediscrim engine details`

`discrim_regularized` *Regularized discriminant analysis*

Description

`discrim_regularized()` defines a model that estimates a multivariate distribution for the predictors separately for the data in each class. The structure of the model can be LDA, QDA, or some amalgam of the two. Bayes' theorem is used to compute the probability of each class, given the predictor values. This function can fit classification models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `klaR`12

1 The default engine. 2 Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
discrim_regularized(
  mode = "classification",
  frac_common_cov = NULL,
  frac_identity = NULL,
  engine = "klaR"
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`frac_common_cov, frac_identity`	Numeric values between zero and one.
`engine`	A single character string specifying what computational engine to use for fitting.

Details

There are many ways of regularizing models. For example, one form of regularization is to penalize model parameters. Similarly, the classic James–Stein regularization approach shrinks the model structure to a less complex form.

The model fits a very specific type of regularized model by Friedman (1989) that uses two types of regularization. One modulates how class-specific the covariance matrix should be. This allows the model to balance between LDA and QDA. The second regularization component shrinks the covariance matrix towards the identity matrix.

For the penalization approach, `discrim_linear()` with a `mda` engine can be used. Other regularization methods can be used with `discrim_linear()` and `discrim_quad()` can used via the `sparsediscrim` engine for those functions.

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
discrim_regularized(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

Friedman, J (1989). Regularized Discriminant Analysis. *Journal of the American Statistical Association*, 84, 165-175.

See Also

`fit()`, `set_engine()`, `update()`, `klaR engine details`

`extract-parsnip` *Extract elements of a parsnip model object*

Description

These functions extract various elements from a parsnip object. If they do not exist yet, an error is thrown.

- `extract_spec_parsnip()` returns the parsnip model specification.
- `extract_fit_engine()` returns the engine specific fit embedded within a parsnip model fit. For example, when using `linear_reg()` with the `"lm"` engine, this returns the underlying `lm` object.
- `extract_parameter_dials()` returns a single dials parameter object.
- `extract_parameter_set_dials()` returns a set of dials parameter objects.
- `extract_fit_time()` returns a tibble with fit times. The fit times correspond to the time for the parsnip engine to fit and do not include other portions of the elapsed time in `fit.model_spec()`.

Usage

```
## S3 method for class 'model_fit'
extract_spec_parsnip(x, ...)

## S3 method for class 'model_fit'
extract_fit_engine(x, ...)

## S3 method for class 'model_spec'
extract_parameter_set_dials(x, ...)

## S3 method for class 'model_spec'
extract_parameter_dials(x, parameter, ...)

## S3 method for class 'model_fit'
extract_fit_time(x, summarize = TRUE, ...)
```

Arguments

`x`	A parsnip `model_fit` object or a parsnip `model_spec` object.
`...`	Not currently used.
`parameter`	A single string for the parameter ID.
`summarize`	A logical for whether the elapsed fit time should be returned as a single row or multiple rows. Doesn't support `FALSE` for parsnip models.

Details

Extracting the underlying engine fit can be helpful for describing the model (via `print()`, `summary()`, `plot()`, etc.) or for variable importance/explainers.

However, users should not invoke the `predict()` method on an extracted model. There may be preprocessing operations that parsnip has executed on the data prior to giving it to the model. Bypassing these can lead to errors or silently generating incorrect predictions.

Good:

```
parsnip_fit %>% predict(new_data)
```

Bad:

```
parsnip_fit %>% extract_fit_engine() %>% predict(new_data)
```

Value

The extracted value from the parsnip object, `x`, as described in the description section.

Examples

```
lm_spec <- linear_reg() %>% set_engine("lm")
lm_fit <- fit(lm_spec, mpg ~ ., data = mtcars)

lm_spec
extract_spec_parsnip(lm_fit)

extract_fit_engine(lm_fit)
lm(mpg ~ ., data = mtcars)
```

`fit.model_spec` *Fit a Model Specification to a Dataset*

Description

`fit()` and `fit_xy()` take a model specification, translate the required code by substituting arguments, and execute the model fit routine.

Usage

```
## S3 method for class 'model_spec'
fit(
  object,
  formula,
  data,
  case_weights = NULL,
  control = control_parsnip(),
  ...
)

## S3 method for class 'model_spec'
fit_xy(object, x, y, case_weights = NULL, control = control_parsnip(), ...)
```

Arguments

`object`	An object of class `model_spec` that has a chosen engine (via `set_engine()`).
`formula`	An object of class `formula` (or one that can be coerced to that class): a symbolic description of the model to be fitted.
`data`	Optional, depending on the interface (see Details below). A data frame containing all relevant variables (e.g. outcome(s), predictors, case weights, etc). Note: when needed, a *named argument* should be used.
`case_weights`	An optional classed vector of numeric case weights. This must return `TRUE` when `hardhat::is_case_weights()` is run on it. See `hardhat::frequency_weights()` and `hardhat::importance_weights()` for examples.
`control`	A named list with elements `verbosity` and `catch`. See `control_parsnip()`.
`...`	Not currently used; values passed here will be ignored. Other options required to fit the model should be passed using `set_engine()`.
`x`	A matrix, sparse matrix, or data frame of predictors. Only some models have support for sparse matrix input. See `parsnip::get_encoding()` for details. `x` should have column names.
`y`	A vector, matrix or data frame of outcome data.

Details

`fit()` and `fit_xy()` substitute the current arguments in the model specification into the computational engine's code, check them for validity, then fit the model using the data and the engine-specific code. Different model functions have different interfaces (e.g. formula or `x/y`) and these functions translate between the interface used when `fit()` or `fit_xy()` was invoked and the one required by the underlying model.

When possible, these functions attempt to avoid making copies of the data. For example, if the underlying model uses a formula and `fit()` is invoked, the original data are references when the model is fit. However, if the underlying model uses something else, such as `x/y`, the formula is evaluated and the data are converted to the required format. In this case, any calls in the resulting model objects reference the temporary objects used to fit the model.

If the model engine has not been set, the model's default engine will be used (as discussed on each model page). If the `verbosity` option of `control_parsnip()` is greater than zero, a warning will be produced.

If you would like to use an alternative method for generating contrasts when supplying a formula to `fit()`, set the global option `contrasts` to your preferred method. For example, you might set it to: `options(contrasts = c(unordered = "contr.helmert", ordered = "contr.poly"))`. See the help page for `stats::contr.treatment()` for more possible contrast types.

For models with `"censored regression"` modes, an additional computation is executed and saved in the parsnip object. The `censor_probs` element contains a "reverse Kaplan-Meier" curve that models the probability of censoring. This may be used later to compute inverse probability censoring weights for performance measures.

Sparse data is supported, with the use of the `x` argument in `fit_xy()`. See `allow_sparse_x` column of `get_encoding()` for sparse input compatibility.

Value

A `model_fit` object that contains several elements:

- `lvl`: If the outcome is a factor, this contains the factor levels at the time of model fitting.
- `spec`: The model specification object (`object` in the call to `fit`)
- `fit`: when the model is executed without error, this is the model object. Otherwise, it is a `try-error` object with the error message.
- `preproc`: any objects needed to convert between a formula and non-formula interface (such as the `terms` object)

The return value will also have a class related to the fitted model (e.g. `"_glm"`) before the base class of `"model_fit"`.

See Also

`set_engine()`, `control_parsnip()`, `model_spec`, `model_fit`

Examples

```
# Although `glm()` only has a formula interface, different
# methods for specifying the model can be used

library(dplyr)
library(modeldata)
data("lending_club")

lr_mod <- logistic_reg()

using_formula <-
  lr_mod %>%
  set_engine("glm") %>%
  fit(Class ~ funded_amnt + int_rate, data = lending_club)

using_xy <-
  lr_mod %>%
   set_engine("glm") %>%
  fit_xy(x = lending_club[, c("funded_amnt", "int_rate")],
         y = lending_club$Class)

using_formula
using_xy
```

`gen_additive_mod` *Generalized additive models (GAMs)*

Description

`gen_additive_mod()` defines a model that can use smoothed functions of numeric predictors in a generalized linear model. This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `mgcv`1

1 The default engine.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
gen_additive_mod(
  mode = "unknown",
  select_features = NULL,
  adjust_deg_free = NULL,
  engine = "mgcv"
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`select_features`	`TRUE` or `FALSE`. If `TRUE`, the model has the ability to eliminate a predictor (via penalization). Increasing `adjust_deg_free` will increase the likelihood of removing predictors.
`adjust_deg_free`	If `select_features = TRUE`, then acts as a multiplier for smoothness. Increase this beyond 1 to produce smoother models.
`engine`	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
gen_additive_mod(argument = !!value)
```

References

https://www.tidymodels.org, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `mgcv engine details`

Examples

```
show_engines("gen_additive_mod")

gen_additive_mod()
```

`glance.model_fit` *Construct a single row summary "glance" of a model, fit, or other object*

Description

This method glances the model in a parsnip model object, if it exists.

Usage

```
## S3 method for class 'model_fit'
glance(x, ...)
```

Arguments

`x`	model or other R object to convert to single-row data frame
`...`	other arguments passed to methods

Value

a tibble

glm_grouped — *Fit a grouped binomial outcome from a data set with case weights*

Description

`stats::glm()` assumes that a tabular data set with case weights corresponds to "different observations have different dispersions" (see `?glm`).

In some cases, the case weights reflect that the same covariate pattern was observed multiple times (i.e., *frequency weights*). In this case, `stats::glm()` expects the data to be formatted as the number of events for each factor level so that the outcome can be given to the formula as `cbind(events_1, events_2)`.

`glm_grouped()` converts data with integer case weights to the expected "number of events" format for binomial data.

Usage

```
glm_grouped(formula, data, weights, ...)
```

Arguments

`formula`	A formula object with one outcome that is a two-level factors.
`data`	A data frame with the outcomes and predictors (but not case weights).
`weights`	An integer vector of weights whose length is the same as the number of rows in `data`. If it is a non-integer numeric, it will be converted to integer (with a warning).
`...`	Options to pass to `stats::glm()`. If `family` is not set, it will automatically be assigned the basic binomial family.

Value

A object produced by `stats::glm()`.

Examples

```
#-------------------------------------------------------------------------------
# The same data set formatted three ways

# First with basic case weights that, from ?glm, are used inappropriately.
ucb_weighted <- as.data.frame(UCBAdmissions)
ucb_weighted$Freq <- as.integer(ucb_weighted$Freq)
head(ucb_weighted)
nrow(ucb_weighted)

# Format when yes/no data are in individual rows (probably still inappropriate)
library(tidyr)
ucb_long <- uncount(ucb_weighted, Freq)
head(ucb_long)
```

```
nrow(ucb_long)

# Format where the outcome is formatted as number of events
ucb_events <-
  ucb_weighted %>%
  tidyr::pivot_wider(
    id_cols = c(Gender, Dept),
    names_from = Admit,
    values_from = Freq,
    values_fill = 0L
  )
head(ucb_events)
nrow(ucb_events)

#------------------------------------------------------------------------------
# Different model fits

# Treat data as separate Bernoulli data:
glm(Admit ~ Gender + Dept, data = ucb_long, family = binomial)

# Weights produce the same statistics
glm(
  Admit ~ Gender + Dept,
  data = ucb_weighted,
  family = binomial,
  weights = ucb_weighted$Freq
)

# Data as binomial "x events out of n trials" format. Note that, to get the same
# coefficients, the order of the levels must be reversed.
glm(
  cbind(Rejected, Admitted) ~ Gender + Dept,
  data = ucb_events,
  family = binomial
)

# The new function that starts with frequency weights and gets the correct place:
glm_grouped(Admit ~ Gender + Dept, data = ucb_weighted, weights = ucb_weighted$Freq)
```

`linear_reg` *Linear regression*

Description

`linear_reg()` defines a model that can predict numeric values from predictors using a linear function. This function can fit regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- lm1
- brulee
- gee2
- glm
- glmer2
- glmnet
- gls2
- h2o2
- keras
- lme2
- lmer2
- spark
- stan
- stan_glmer2

1 The default engine. 2 Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
linear_reg(mode = "regression", engine = "lm", penalty = NULL, mixture = NULL)
```

Arguments

`mode`	A single character string for the type of model. The only possible value for this model is "regression".
`engine`	A single character string specifying what computational engine to use for fitting. Possible engines are listed below. The default for this model is `"lm"`.
`penalty`	A non-negative number representing the total amount of regularization (specific engines only).
`mixture`	A number between zero and one (inclusive) denoting the proportion of L1 regularization (i.e. lasso) in the model. • `mixture = 1` specifies a pure lasso model, • `mixture = 0` specifies a ridge regression model, and • `0 < mixture < 1` specifies an elastic net model, interpolating lasso and ridge. Available for specific engines only.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
linear_reg(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `lm engine details`, `brulee engine details`, `gee engine details`, `glm engine details`, `glmer engine details`, `glmnet engine details`, `gls engine details`, `h2o engine details`, `keras engine details`, `lme engine details`, `lmer engine details`, `spark engine details`, `stan engine details`, `stan_glmer engine details`

Examples

```
show_engines("linear_reg")

linear_reg()
```

logistic_reg *Logistic regression*

Description

`logistic_reg()` defines a generalized linear model for binary outcomes. A linear combination of the predictors is used to model the log odds of an event. This function can fit classification models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `glm`1
- `brulee`
- `gee`2
- `glmer`2
- `glmnet`
- `h2o`2

- keras
- LiblineaR
- spark
- stan
- stan_glmer2

1 The default engine. 2 Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at https://www.tidymodels.org/.

Usage

```
logistic_reg(
  mode = "classification",
  engine = "glm",
  penalty = NULL,
  mixture = NULL
)
```

Arguments

mode	A single character string for the type of model. The only possible value for this model is "classification".
engine	A single character string specifying what computational engine to use for fitting. Possible engines are listed below. The default for this model is "glm".
penalty	A non-negative number representing the total amount of regularization (specific engines only). For keras models, this corresponds to purely L2 regularization (aka weight decay) while the other models can be either or a combination of L1 and L2 (depending on the value of mixture).
mixture	A number between zero and one (inclusive) giving the proportion of L1 regularization (i.e. lasso) in the model. • mixture = 1 specifies a pure lasso model, • mixture = 0 specifies a ridge regression model, and • 0 < mixture < 1 specifies an elastic net model, interpolating lasso and ridge. Available for specific engines only. For LiblineaR models, mixture must be exactly 1 or 0 only.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See set_engine() for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the fit() function is used with the data.

Each of the arguments in this function other than mode and engine are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
logistic_reg(argument = !!value)
```

This model fits a classification model for binary outcomes; for multiclass outcomes, see `multinom_reg()`.

References

https://www.tidymodels.org, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, glm engine details, brulee engine details, gee engine details, glmer engine details, glmnet engine details, h2o engine details, keras engine details, LiblineaR engine details, spark engine details, stan engine details, stan_glmer engine details

Examples

```
show_engines("logistic_reg")

logistic_reg()
```

`mars` *Multivariate adaptive regression splines (MARS)*

Description

`mars()` defines a generalized linear model that uses artificial features for some predictors. These features resemble hinge functions and the result is a model that is a segmented regression in small dimensions. This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- earth1

1 The default engine.

More information on how **parsnip** is used for modeling is at https://www.tidymodels.org/.

Usage

```
mars(
  mode = "unknown",
  engine = "earth",
  num_terms = NULL,
  prod_degree = NULL,
  prune_method = NULL
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`engine`	A single character string specifying what computational engine to use for fitting.
`num_terms`	The number of features that will be retained in the final model, including the intercept.
`prod_degree`	The highest possible interaction degree.
`prune_method`	The pruning method.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
mars(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `earth engine details`

Examples

```
show_engines("mars")

mars(mode = "regression", num_terms = 5)
```

`max_mtry_formula` *Determine largest value of mtry from formula. This function potentially caps the value of* `mtry` *based on a formula and data set. This is a safe approach for survival and/or multivariate models.*

Description

Determine largest value of mtry from formula. This function potentially caps the value of `mtry` based on a formula and data set. This is a safe approach for survival and/or multivariate models.

Usage

```
max_mtry_formula(mtry, formula, data)
```

Arguments

`mtry`	An initial value of `mtry` (which may be too large).
`formula`	A model formula.
`data`	The training set (data frame).

Value

A value for `mtry`.

Examples

```
# should be 9
max_mtry_formula(200, cbind(wt, mpg) ~ ., data = mtcars)
```

`maybe_matrix` *Fuzzy conversions*

Description

These are substitutes for `as.matrix()` and `as.data.frame()` that leave a sparse matrix as-is.

Usage

```
maybe_matrix(x)

maybe_data_frame(x)
```

Arguments

`x`	A data frame, matrix, or sparse matrix.

Value

A data frame, matrix, or sparse matrix.

min_cols *Execution-time data dimension checks*

Description

For some tuning parameters, the range of values depend on the data dimensions (e.g. `mtry`). Some packages will fail if the parameter values are outside of these ranges. Since the model might receive resampled versions of the data, these ranges can't be set prior to the point where the model is fit. These functions check the possible range of the data and adjust them if needed (with a warning).

Usage

```
min_cols(num_cols, source)

min_rows(num_rows, source, offset = 0)
```

Arguments

`num_cols`, `num_rows`	The parameter value requested by the user.
`source`	A data frame for the data to be used in the fit. If the source is named "data", it is assumed that one column of the data corresponds to an outcome (and is subtracted off).
`offset`	A number subtracted off of the number of rows available in the data.

Value

An integer (and perhaps a warning).

Examples

```
nearest_neighbor(neighbors= 100) %>%
  set_engine("kknn") %>%
  set_mode("regression") %>%
  translate()

library(ranger)
rand_forest(mtry = 2, min_n = 100, trees = 3) %>%
  set_engine("ranger") %>%
  set_mode("regression") %>%
  fit(mpg ~ ., data = mtcars)
```

mlp *Single layer neural network*

Description

`mlp()` defines a multilayer perceptron model (a.k.a. a single layer, feed-forward neural network). This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `nnet`1
- `brulee`
- `h2o`2
- `keras`

1 The default engine. 2 Requires a parsnip extension package for classification and regression.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
mlp(
  mode = "unknown",
  engine = "nnet",
  hidden_units = NULL,
  penalty = NULL,
  dropout = NULL,
  epochs = NULL,
  activation = NULL,
  learn_rate = NULL
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`engine`	A single character string specifying what computational engine to use for fitting.
`hidden_units`	An integer for the number of units in the hidden model.
`penalty`	A non-negative numeric value for the amount of weight decay.
`dropout`	A number between 0 (inclusive) and 1 denoting the proportion of model parameters randomly set to zero during model training.
`epochs`	An integer for the number of training iterations.
`activation`	A single character string denoting the type of relationship between the original predictors and the hidden unit layer. The activation function between the hidden and output layers is automatically set to either "linear" or "softmax" depending on the type of outcome. Possible values depend on the engine being used.

`learn_rate`	A number for the rate at which the boosting algorithm adapts from iteration-to-iteration (specific engines only). This is sometimes referred to as the shrinkage parameter.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
mlp(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `nnet engine details`, `brulee engine details`, `h2o engine details`, `keras engine details`

Examples

```
show_engines("mlp")

mlp(mode = "classification", penalty = 0.01)
```

`model_fit` *Model Fit Objects*

Description

Model fits are trained model specifications that are ready to predict on new data. Model fits have class `model_fit` and, usually, a subclass referring to the engine used to fit the model.

Details

An object with class `"model_fit"` is a container for information about a model that has been fit to the data.

The main elements of the object are:

- `lvl`: A vector of factor levels when the outcome is a factor. This is `NULL` when the outcome is not a factor vector.

- `spec`: A `model_spec` object.
- `fit`: The object produced by the fitting function.
- `preproc`: This contains any data-specific information required to process new a sample point for prediction. For example, if the underlying model function requires arguments `x` and `y` and the user passed a formula to `fit`, the `preproc` object would contain items such as the terms object and so on. When no information is required, this is `NA`.

As discussed in the documentation for `model_spec`, the original arguments to the specification are saved as quosures. These are evaluated for the `model_fit` object prior to fitting. If the resulting model object prints its call, any user-defined options are shown in the call preceded by a tilde (see the example below). This is a result of the use of quosures in the specification.

This class and structure is the basis for how **parsnip** stores model objects after seeing the data and applying a model.

Examples

```
# Keep the `x` matrix if the data are not too big.
spec_obj <-
  linear_reg() %>%
  set_engine("lm", x = ifelse(.obs() < 500, TRUE, FALSE))
spec_obj

fit_obj <- fit(spec_obj, mpg ~ ., data = mtcars)
fit_obj

nrow(fit_obj$fit$x)
```

`model_formula` *Formulas with special terms in tidymodels*

Description

In R, formulas provide a compact, symbolic notation to specify model terms. Many modeling functions in R make use of "specials", or nonstandard notations used in formulas. Specials are defined and handled as a special case by a given modeling package. For example, the mgcv package, which provides support for generalized additive models in R, defines a function `s()` to be in-lined into formulas. It can be used like so:

```
mgcv::gam(mpg ~ wt + s(disp, k = 5), data = mtcars)
```

In this example, the `s()` special defines a smoothing term that the mgcv package knows to look for when preprocessing model input.

The parsnip package can handle most specials without issue. The analogous code for specifying this generalized additive model with the parsnip "mgcv" engine looks like:

```
gen_additive_mod() %>%
  set_mode("regression") %>%
  set_engine("mgcv") %>%
  fit(mpg ~ wt + s(disp, k = 5), data = mtcars)
```

However, parsnip is often used in conjunction with the greater tidymodels package ecosystem, which defines its own pre-processing infrastructure and functionality via packages like hardhat and recipes. The specials defined in many modeling packages introduce conflicts with that infrastructure.

To support specials while also maintaining consistent syntax elsewhere in the ecosystem, **tidymodels delineates between two types of formulas: preprocessing formulas and model formulas**. Preprocessing formulas specify the input variables, while model formulas determine the model structure.

Example

To create the preprocessing formula from the model formula, just remove the specials, retaining references to input variables themselves. For example:

```
model_formula <- mpg ~ wt + s(disp, k = 5)
preproc_formula <- mpg ~ wt + disp
```

- **With parsnip,** use the model formula:

  ```
  model_spec <-
    gen_additive_mod() %>%
    set_mode("regression") %>%
    set_engine("mgcv")

  model_spec %>%
    fit(model_formula, data = mtcars)
  ```

- **With recipes**, use the preprocessing formula only:

  ```
  library(recipes)

  recipe(preproc_formula, mtcars)
  ```

  The recipes package supplies a large variety of preprocessing techniques that may replace the need for specials altogether, in some cases.

- **With workflows,** use the preprocessing formula everywhere, but pass the model formula to the `formula` argument in `add_model()`:

  ```
  library(workflows)

  wflow <-
    workflow() %>%
    add_formula(preproc_formula) %>%
    add_model(model_spec, formula = model_formula)

  fit(wflow, data = mtcars)
  ```

The workflow will then pass the model formula to parsnip, using the preprocessor formula elsewhere. We would still use the preprocessing formula if we had added a recipe preprocessor using `add_recipe()` instead a formula via `add_formula()`.

`model_spec` *Model Specifications*

Description

The parsnip package splits the process of fitting models into two steps:

1. Specify how a model will be fit using a *model specification*
2. Fit a model using the model specification

This is a different approach to many other model interfaces in R, like `lm()`, where both the specification of the model and the fitting happens in one function call. Splitting the process into two steps allows users to iteratively define model specifications throughout the model development process.

This intermediate object that defines how the model will be fit is called a *model specification* and has class `model_spec`. Model type functions, like `linear_reg()` or `boost_tree()`, return `model_spec` objects.

Fitted model objects, resulting from passing a `model_spec` to fit() or fit_xy, have class `model_fit`, and contain the original `model_spec` objects inside them. See ?model_fit for more on that object type, and ?extract_spec_parsnip to extract `model_spec`s from `model_fit`s.

Details

An object with class `"model_spec"` is a container for information about a model that will be fit.

The main elements of the object are:

- `args`: A vector of the main arguments for the model. The names of these arguments may be different from their counterparts n the underlying model function. For example, for a `glmnet` model, the argument name for the amount of the penalty is called "penalty" instead of "lambda" to make it more general and usable across different types of models (and to not be specific to a particular model function). The elements of `args` can `tune()` with the use of the tune package. For more information see `https://www.tidymodels.org/start/tuning/`. If left to their defaults (`NULL`), the arguments will use the underlying model functions default value. As discussed below, the arguments in `args` are captured as quosures and are not immediately executed.
- `...`: Optional model-function-specific parameters. As with `args`, these will be quosures and can be `tune()`.
- `mode`: The type of model, such as "regression" or "classification". Other modes will be added once the package adds more functionality.
- `method`: This is a slot that is filled in later by the model's constructor function. It generally contains lists of information that are used to create the fit and prediction code as well as required packages and similar data.

- `engine`: This character string declares exactly what software will be used. It can be a package name or a technology type.

This class and structure is the basis for how parsnip stores model objects prior to seeing the data.

Argument Details

An important detail to understand when creating model specifications is that they are intended to be functionally independent of the data. While it is true that some tuning parameters are *data dependent*, the model specification does not interact with the data at all.

For example, most R functions immediately evaluate their arguments. For example, when calling `mean(dat_vec)`, the object `dat_vec` is immediately evaluated inside of the function.

parsnip model functions do not do this. For example, using

```
rand_forest(mtry = ncol(mtcars) - 1)
```

does not execute `ncol(mtcars) - 1` when creating the specification. This can be seen in the output:

```
> rand_forest(mtry = ncol(mtcars) - 1)
Random Forest Model Specification (unknown)

Main Arguments:
  mtry = ncol(mtcars) - 1
```

The model functions save the argument *expressions* and their associated environments (a.k.a. a quosure) to be evaluated later when either `fit.model_spec()` or `fit_xy.model_spec()` are called with the actual data.

The consequence of this strategy is that any data required to get the parameter values must be available when the model is fit. The two main ways that this can fail is if:

1. The data have been modified between the creation of the model specification and when the model fit function is invoked.
2. If the model specification is saved and loaded into a new session where those same data objects do not exist.

The best way to avoid these issues is to not reference any data objects in the global environment but to use data descriptors such as `.cols()`. Another way of writing the previous specification is

```
rand_forest(mtry = .cols() - 1)
```

This is not dependent on any specific data object and is evaluated immediately before the model fitting process begins.

One less advantageous approach to solving this issue is to use quasiquotation. This would insert the actual R object into the model specification and might be the best idea when the data object is small. For example, using

```
rand_forest(mtry = ncol(!!mtcars) - 1)
```

would work (and be reproducible between sessions) but embeds the entire mtcars data set into the `mtry` expression:

```
> rand_forest(mtry = ncol(!!mtcars) - 1)
Random Forest Model Specification (unknown)

Main Arguments:
  mtry = ncol(structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5, <snip>
```

However, if there were an object with the number of columns in it, this wouldn't be too bad:

```
> mtry_val <- ncol(mtcars) - 1
> mtry_val
[1] 10
> rand_forest(mtry = !!mtry_val)
Random Forest Model Specification (unknown)

Main Arguments:
  mtry = 10
```

More information on quosures and quasiquotation can be found at `https://adv-r.hadley.nz/quasiquotation.html`.

`multinom_reg` *Multinomial regression*

Description

`multinom_reg()` defines a model that uses linear predictors to predict multiclass data using the multinomial distribution. This function can fit classification models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `nnet`1
- `brulee`
- `glmnet`
- `h2o`2
- `keras`
- `spark`

1 The default engine. 2 Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
multinom_reg(
  mode = "classification",
  engine = "nnet",
  penalty = NULL,
  mixture = NULL
)
```

Arguments

`mode`	A single character string for the type of model. The only possible value for this model is "classification".
`engine`	A single character string specifying what computational engine to use for fitting. Possible engines are listed below. The default for this model is `"nnet"`.
`penalty`	A non-negative number representing the total amount of regularization (specific engines only). For `keras` models, this corresponds to purely L2 regularization (aka weight decay) while the other models can be a combination of L1 and L2 (depending on the value of `mixture`).
`mixture`	A number between zero and one (inclusive) giving the proportion of L1 regularization (i.e. lasso) in the model. • `mixture = 1` specifies a pure lasso model, • `mixture = 0` specifies a ridge regression model, and • `0 < mixture < 1` specifies an elastic net model, interpolating lasso and ridge. Available for specific engines only.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
multinom_reg(argument = !!value)
```

This model fits a classification model for multiclass outcomes; for binary outcomes, see `logistic_reg()`.

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

fit(), set_engine(), update(), nnet engine details, brulee engine details, glmnet engine details, h2o engine details, keras engine details, spark engine details

Examples

```
show_engines("multinom_reg")

multinom_reg()
```

`multi_predict` *Model predictions across many sub-models*

Description

For some models, predictions can be made on sub-models in the model object.

Usage

```
multi_predict(object, ...)

## Default S3 method:
multi_predict(object, ...)

## S3 method for class '`_xgb.Booster`'
multi_predict(object, new_data, type = NULL, trees = NULL, ...)

## S3 method for class '`_C5.0`'
multi_predict(object, new_data, type = NULL, trees = NULL, ...)

## S3 method for class '`_elnet`'
multi_predict(object, new_data, type = NULL, penalty = NULL, ...)

## S3 method for class '`_lognet`'
multi_predict(object, new_data, type = NULL, penalty = NULL, ...)

## S3 method for class '`_multnet`'
multi_predict(object, new_data, type = NULL, penalty = NULL, ...)

## S3 method for class '`_glmnetfit`'
multi_predict(object, new_data, type = NULL, penalty = NULL, ...)

## S3 method for class '`_earth`'
multi_predict(object, new_data, type = NULL, num_terms = NULL, ...)

## S3 method for class '`_torch_mlp`'
```

```
multi_predict(object, new_data, type = NULL, epochs = NULL, ...)

## S3 method for class '`_train.kknn`'
multi_predict(object, new_data, type = NULL, neighbors = NULL, ...)
```

Arguments

`object`	A model fit.
`...`	Optional arguments to pass to `predict.model_fit(type = "raw")` such as `type`.
`new_data`	A rectangular data object, such as a data frame.
`type`	A single character value or `NULL`. Possible values are `"numeric"`, `"class"`, `"prob"`, `"conf_int"`, `"pred_int"`, `"quantile"`, or `"raw"`. When `NULL`, `predict()` will choose an appropriate value based on the model's mode.
`trees`	An integer vector for the number of trees in the ensemble.
`penalty`	A numeric vector of penalty values.
`num_terms`	An integer vector for the number of MARS terms to retain.
`epochs`	An integer vector for the number of training epochs.
`neighbors`	An integer vector for the number of nearest neighbors.

Value

A tibble with the same number of rows as the data being predicted. There is a list-column named `.pred` that contains tibbles with multiple rows per sub-model. Note that, within the tibbles, the column names follow the usual standard based on prediction `type` (i.e. `.pred_class` for `type = "class"` and so on).

`naive_Bayes` *Naive Bayes models*

Description

`naive_Bayes()` defines a model that uses Bayes' theorem to compute the probability of each class, given the predictor values. This function can fit classification models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `klaR`¹²
- `h2o`²
- `naivebayes`²

¹ The default engine. ² Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
naive_Bayes(
  mode = "classification",
  smoothness = NULL,
  Laplace = NULL,
  engine = "klaR"
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`smoothness`	An non-negative number representing the the relative smoothness of the class boundary. Smaller examples result in model flexible boundaries and larger values generate class boundaries that are less adaptable
`Laplace`	A non-negative value for the Laplace correction to smoothing low-frequency counts.
`engine`	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
naive_Bayes(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `klaR engine details`, `h2o engine details`, `naivebayes engine details`

`nearest_neighbor` *K-nearest neighbors*

Description

`nearest_neighbor()` defines a model that uses the `K` most similar data points from the training set to predict new samples. This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `kknn`1

1 The default engine.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
nearest_neighbor(
  mode = "unknown",
  engine = "kknn",
  neighbors = NULL,
  weight_func = NULL,
  dist_power = NULL
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`engine`	A single character string specifying what computational engine to use for fitting.
`neighbors`	A single integer for the number of neighbors to consider (often called `k`). For **kknn**, a value of 5 is used if `neighbors` is not specified.
`weight_func`	A *single* character for the type of kernel function used to weight distances between samples. Valid choices are: `"rectangular"`, `"triangular"`, `"epanechnikov"`, `"biweight"`, `"triweight"`, `"cos"`, `"inv"`, `"gaussian"`, `"rank"`, or `"optimal"`.
`dist_power`	A single number for the parameter used in calculating Minkowski distance.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
nearest_neighbor(argument = !!value)
```

References

https://www.tidymodels.org, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `kknn engine details`

Examples

```
show_engines("nearest_neighbor")

nearest_neighbor(neighbors = 11)
```

`null_model` *Null model*

Description

`null_model()` defines a simple, non-informative model. It doesn't have any main arguments. This function can fit classification and regression models.

Usage

```
null_model(mode = "classification", engine = "parsnip")
```

Arguments

`mode`	A single character string for the type of model. The only possible values for this model are `"regression"` and `"classification"`.
`engine`	A single character string specifying what computational engine to use for fitting. Possible engines are listed below. The default for this model is `"parsnip"`.

Engine Details

Engines may have pre-set default arguments when executing the model fit call. For this type of model, the template of the fit calls are below:

parsnip:

```
null_model() %>%
  set_engine("parsnip") %>%
  set_mode("regression") %>%
  translate()
```

```
## Null Model Specification (regression)
##
## Computational engine: parsnip
##
## Model fit template:
## parsnip::nullmodel(x = missing_arg(), y = missing_arg())

null_model() %>%
  set_engine("parsnip") %>%
  set_mode("classification") %>%
  translate()

## Null Model Specification (classification)
##
## Computational engine: parsnip
##
## Model fit template:
## parsnip::nullmodel(x = missing_arg(), y = missing_arg())
```

See Also

`fit.model_spec()`

Examples

```
null_model(mode = "regression")
```

---

parsnip_addin *Start an RStudio Addin that can write model specifications*

Description

`parsnip_addin()` starts a process in the RStudio IDE Viewer window that allows users to write code for parsnip model specifications from various R packages. The new code is written to the current document at the location of the cursor.

Usage

```
parsnip_addin()
```

pls *Partial least squares (PLS)*

Description

pls() defines a partial least squares model that uses latent variables to model the data. It is similar to a supervised version of principal component. This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- mixOmics12

1 The default engine. 2 Requires a parsnip extension package for classification and regression.

More information on how **parsnip** is used for modeling is at https://www.tidymodels.org/.

Usage

```
pls(
  mode = "unknown",
  predictor_prop = NULL,
  num_comp = NULL,
  engine = "mixOmics"
)
```

Arguments

mode	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
predictor_prop	The maximum proportion of original predictors that can have *non-zero* coefficients for each PLS component (via regularization). This value is used for all PLS components for X.
num_comp	The number of PLS components to retain.
engine	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See set_engine() for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the fit() function is used with the data.

Each of the arguments in this function other than mode and engine are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
pls(argument = !!value)
```

References

https://www.tidymodels.org, *Tidy Modeling with R*, searchable table of parsnip models

See Also

fit(), set_engine(), update(), mixOmics engine details

---

poisson_reg — *Poisson regression models*

Description

poisson_reg() defines a generalized linear model for count data that follow a Poisson distribution. This function can fit regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- glm12
- gee2
- glmer2
- glmnet2
- h2o2
- hurdle2
- stan2
- stan_glmer2
- zeroinfl2

1 The default engine. 2 Requires a parsnip extension package.

More information on how **parsnip** is used for modeling is at https://www.tidymodels.org/.

Usage

```
poisson_reg(
  mode = "regression",
  penalty = NULL,
  mixture = NULL,
  engine = "glm"
)
```

Arguments

`mode`	A single character string for the type of model. The only possible value for this model is "regression".
`penalty`	A non-negative number representing the total amount of regularization (`glmnet` only).
`mixture`	A number between zero and one (inclusive) giving the proportion of L1 regularization (i.e. lasso) in the model. • `mixture = 1` specifies a pure lasso model, • `mixture = 0` specifies a ridge regression model, and • `0 < mixture < 1` specifies an elastic net model, interpolating lasso and ridge. Available for `glmnet` and `spark` only.
`engine`	A single character string specifying what computational engine to use for fitting.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
poisson_reg(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `glm engine details`, `gee engine details`, `glmer engine details`, `glmnet engine details`, `h2o engine details`, `hurdle engine details`, `stan engine details`, `stan_glmer engine details`, `zeroinfl engine details`

`rand_forest` *Random forest*

Description

`rand_forest()` defines a model that creates a large number of decision trees, each independent of the others. The final prediction uses all predictions from the individual trees and combines them. This function can fit classification, regression, and censored regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `ranger`1
- `aorsf`2
- `h2o`2
- `partykit`2
- `randomForest`
- `spark`

1 The default engine. 2 Requires a parsnip extension package for censored regression, classification, and regression.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
rand_forest(
  mode = "unknown",
  engine = "ranger",
  mtry = NULL,
  trees = NULL,
  min_n = NULL
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", "classification", or "censored regression".
`engine`	A single character string specifying what computational engine to use for fitting.
`mtry`	An integer for the number of predictors that will be randomly sampled at each split when creating the tree models.
`trees`	An integer for the number of trees contained in the ensemble.
`min_n`	An integer for the minimum number of data points in a node that are required for the node to be split further.

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
rand_forest(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `ranger` engine details, `aorsf` engine details, `h2o` engine details, `partykit` engine details, `randomForest` engine details, `spark` engine details

Examples

```
show_engines("rand_forest")

rand_forest(mode = "classification", trees = 2000)
```

`repair_call` *Repair a model call object*

Description

When the user passes a formula to `fit()` *and* the underlying model function uses a formula, the call object produced by `fit()` may not be usable by other functions. For example, some arguments may still be quosures and the `data` portion of the call will not correspond to the original data.

Usage

```
repair_call(x, data)
```

Arguments

`x`	A fitted parsnip model. An error will occur if the underlying model does not have a `call` element.
`data`	A data object that is relevant to the call. In most cases, this is the data frame that was given to parsnip for the model fit (i.e., the training set data). The name of this data object is inserted into the call.

Details

`repair_call()` call can adjust the model objects call to be usable by other functions and methods.

Value

A modified `parsnip` fitted model.

Examples

```
fitted_model <-
  linear_reg() %>%
  set_engine("lm", model = TRUE) %>%
  fit(mpg ~ ., data = mtcars)

# In this call, note that `data` is not `mtcars` and the `model = ~TRUE`
# indicates that the `model` argument is an rlang quosure.
fitted_model$fit$call

# All better:
repair_call(fitted_model, mtcars)$fit$call
```

---

required_pkgs.model_spec

Determine required packages for a model

Description

Determine required packages for a model

Usage

```
## S3 method for class 'model_spec'
required_pkgs(x, infra = TRUE, ...)

## S3 method for class 'model_fit'
required_pkgs(x, infra = TRUE, ...)
```

Arguments

`x`	A model specification or fit.
`infra`	Should parsnip itself be included in the result?
`...`	Not used.

Value

A character vector

Examples

```
should_fail <- try(required_pkgs(linear_reg(engine = NULL)), silent = TRUE)
should_fail

linear_reg() %>%
  set_engine("glmnet") %>%
  required_pkgs()

linear_reg() %>%
  set_engine("glmnet") %>%
  required_pkgs(infra = FALSE)

linear_reg() %>%
  set_engine("lm") %>%
  fit(mpg ~ ., data = mtcars) %>%
  required_pkgs()
```

`req_pkgs` *Determine required packages for a model*

Description

[Deprecated]

Usage

```
req_pkgs(x, ...)
```

Arguments

`x`	A model specification or fit.
`...`	Not used.

Details

This function has been deprecated in favor of `required_pkgs()`.

Value

A character string of package names (if any).

rule_fit	*RuleFit models*

Description

`rule_fit()` defines a model that derives simple feature rules from a tree ensemble and uses them as features in a regularized model. This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `xrf`12
- `h2o`2

1 The default engine. 2 Requires a parsnip extension package for classification and regression.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
rule_fit(
  mode = "unknown",
  mtry = NULL,
  trees = NULL,
  min_n = NULL,
  tree_depth = NULL,
  learn_rate = NULL,
  loss_reduction = NULL,
  sample_size = NULL,
  stop_iter = NULL,
  penalty = NULL,
  engine = "xrf"
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`mtry`	A number for the number (or proportion) of predictors that will be randomly sampled at each split when creating the tree models (specific engines only).
`trees`	An integer for the number of trees contained in the ensemble.
`min_n`	An integer for the minimum number of data points in a node that is required for the node to be split further.
`tree_depth`	An integer for the maximum depth of the tree (i.e. number of splits) (specific engines only).
`learn_rate`	A number for the rate at which the boosting algorithm adapts from iteration-to-iteration (specific engines only). This is sometimes referred to as the shrinkage parameter.

`loss_reduction`	A number for the reduction in the loss function required to split further (specific engines only).
`sample_size`	A number for the number (or proportion) of data that is exposed to the fitting routine. For `xgboost`, the sampling is done at each iteration while `C5.0` samples once during training.
`stop_iter`	The number of iterations without improvement before stopping (specific engines only).
`penalty`	L1 regularization parameter.
`engine`	A single character string specifying what computational engine to use for fitting.

Details

The RuleFit model creates a regression model of rules in two stages. The first stage uses a tree-based model that is used to generate a set of rules that can be filtered, modified, and simplified. These rules are then added as predictors to a regularized generalized linear model that can also conduct feature selection during model training.

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
rule_fit(argument = !!value)
```

References

Friedman, J. H., and Popescu, B. E. (2008). "Predictive learning via rule ensembles." *The Annals of Applied Statistics*, 2(3), 916-954.

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`xrf::xrf.formula()`, `fit()`, `set_engine()`, `update()`, `xrf engine details`, `h2o engine details`

Examples

```
show_engines("rule_fit")

rule_fit()
```

`set_args` *Change elements of a model specification*

Description

`set_args()` can be used to modify the arguments of a model specification while `set_mode()` is used to change the model's mode.

Usage

```
set_args(object, ...)

set_mode(object, mode)
```

Arguments

`object`	A model specification.
`...`	One or more named model arguments.
`mode`	A character string for the model type (e.g. "classification" or "regression")

Details

`set_args()` will replace existing values of the arguments.

Value

An updated model object.

Examples

```
rand_forest()

rand_forest() %>%
  set_args(mtry = 3, importance = TRUE) %>%
  set_mode("regression")
```

set_engine — *Declare a computational engine and specific arguments*

Description

`set_engine()` is used to specify which package or system will be used to fit the model, along with any arguments specific to that software.

Usage

```
set_engine(object, engine, ...)
```

Arguments

`object`	A model specification.
`engine`	A character string for the software that should be used to fit the model. This is highly dependent on the type of model (e.g. linear regression, random forest, etc.).
`...`	Any optional arguments associated with the chosen computational engine. These are captured as quosures and can be tuned with `tune()`.

Details

In parsnip,

- the model **type** differentiates basic modeling approaches, such as random forests, logistic regression, linear support vector machines, etc.,
- the **mode** denotes in what kind of modeling context it will be used (most commonly, classification or regression), and
- the computational **engine** indicates how the model is fit, such as with a specific R package implementation or even methods outside of R like Keras or Stan.

Use `show_engines()` to get a list of possible engines for the model of interest.

Modeling functions in parsnip separate model arguments into two categories:

- *Main arguments* are more commonly used and tend to be available across engines. These names are standardized to work with different engines in a consistent way, so you can use the parsnip main argument `trees`, instead of the heterogeneous arguments for this parameter from **ranger** and **randomForest** packages (`num.trees` and `ntree`, respectively). Set these in your model type function, like `rand_forest(trees = 2000)`.
- *Engine arguments* are either specific to a particular engine or used more rarely; there is no change for these argument names from the underlying engine. The `...` argument of `set_engine()` allows any engine-specific argument to be passed directly to the engine fitting function, like `set_engine("ranger", importance = "permutation")`.

Value

An updated model specification.

Examples

```
# First, set main arguments using the standardized names
logistic_reg(penalty = 0.01, mixture = 1/3) %>%
  # Now specify how you want to fit the model with another argument
  set_engine("glmnet", nlambda = 10) %>%
  translate()

# Many models have possible engine-specific arguments
decision_tree(tree_depth = 5) %>%
  set_engine("rpart", parms = list(prior = c(.65,.35))) %>%
  set_mode("classification") %>%
  translate()
```

`show_engines` *Display currently available engines for a model*

Description

The possible engines for a model can depend on what packages are loaded. Some parsnip extension add engines to existing models. For example, the **poissonreg** package adds additional engines for the `poisson_reg()` model and these are not available unless **poissonreg** is loaded.

Usage

```
show_engines(x)
```

Arguments

`x`	The name of a parsnip model (e.g., "linear_reg", "mars", etc.)

Value

A tibble.

Examples

```
show_engines("linear_reg")
```

sparse_data *Using sparse data with parsnip*

Description

You can figure out whether a given model engine supports sparse data by calling `get_encoding("name of model")` and looking at the `allow_sparse_x` column.

Details

Using sparse data for model fitting and prediction shouldn't require any additional configurations. Just pass in a sparse matrix such as dgCMatrix from the `Matrix` package or a sparse tibble from the sparsevctrs package to the data argument of `fit()`, `fit_xy()`, and `predict()`.

Models that don't support sparse data will try to convert to non-sparse data with warnings. If conversion isn't possible, an informative error will be thrown.

svm_linear *Linear support vector machines*

Description

`svm_linear()` defines a support vector machine model. For classification, the model tries to maximize the width of the margin between classes (using a linear class boundary). For regression, the model optimizes a robust loss function that is only affected by very large model residuals and uses a linear fit. This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `LiblineaR`1
- `kernlab`

1 The default engine.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
svm_linear(mode = "unknown", engine = "LiblineaR", cost = NULL, margin = NULL)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`engine`	A single character string specifying what computational engine to use for fitting.
`cost`	A positive number for the cost of predicting a sample within or on the wrong side of the margin
`margin`	A positive number for the epsilon in the SVM insensitive loss function (regression only)

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
svm_linear(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `LiblineaR engine details`, `kernlab engine details`

Examples

```
show_engines("svm_linear")

svm_linear(mode = "classification")
```

`svm_poly` *Polynomial support vector machines*

Description

`svm_poly()` defines a support vector machine model. For classification, the model tries to maximize the width of the margin between classes using a polynomial class boundary. For regression, the model optimizes a robust loss function that is only affected by very large model residuals and uses polynomial functions of the predictors. This function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `kernlab`1

1 The default engine.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
svm_poly(
  mode = "unknown",
  engine = "kernlab",
  cost = NULL,
  degree = NULL,
  scale_factor = NULL,
  margin = NULL
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`engine`	A single character string specifying what computational engine to use for fitting.
`cost`	A positive number for the cost of predicting a sample within or on the wrong side of the margin
`degree`	A positive number for polynomial degree.
`scale_factor`	A positive number for the polynomial scaling factor.
`margin`	A positive number for the epsilon in the SVM insensitive loss function (regression only)

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than `mode` and `engine` are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
svm_poly(argument = !!value)
```

References

`https://www.tidymodels.org`, *Tidy Modeling with R*, searchable table of parsnip models

See Also

`fit()`, `set_engine()`, `update()`, `kernlab engine details`

Examples

```
show_engines("svm_poly")

svm_poly(mode = "classification", degree = 1.2)
```

svm_rbf — *Radial basis function support vector machines*

Description

`svm_rbf()` defines a support vector machine model. For classification, the model tries to maximize the width of the margin between classes using a nonlinear class boundary. For regression, the model optimizes a robust loss function that is only affected by very large model residuals and uses nonlinear functions of the predictors. The function can fit classification and regression models.

There are different ways to fit this model, and the method of estimation is chosen by setting the model *engine*. The engine-specific pages for this model are listed below.

- `kernlab`1

1 The default engine.

More information on how **parsnip** is used for modeling is at `https://www.tidymodels.org/`.

Usage

```
svm_rbf(
  mode = "unknown",
  engine = "kernlab",
  cost = NULL,
  rbf_sigma = NULL,
  margin = NULL
)
```

Arguments

`mode`	A single character string for the prediction outcome mode. Possible values for this model are "unknown", "regression", or "classification".
`engine`	A single character string specifying what computational engine to use for fitting. Possible engines are listed below. The default for this model is `"kernlab"`.
`cost`	A positive number for the cost of predicting a sample within or on the wrong side of the margin
`rbf_sigma`	A positive number for radial basis function.
`margin`	A positive number for the epsilon in the SVM insensitive loss function (regression only)

Details

This function only defines what *type* of model is being fit. Once an engine is specified, the *method* to fit the model is also defined. See `set_engine()` for more on setting the engine, including how to set engine arguments.

The model is not trained or fit until the `fit()` function is used with the data.

Each of the arguments in this function other than mode and engine are captured as quosures. To pass values programmatically, use the injection operator like so:

```
value <- 1
svm_rbf(argument = !!value)
```

References

https://www.tidymodels.org, *Tidy Modeling with R*, searchable table of parsnip models

See Also

fit(), set_engine(), update(), kernlab engine details

Examples

```
show_engines("svm_rbf")

svm_rbf(mode = "classification", rbf_sigma = 0.2)
```

---

tidy.model_fit *Turn a parsnip model object into a tidy tibble*

Description

This method tidies the model in a parsnip model object, if it exists.

Usage

```
## S3 method for class 'model_fit'
tidy(x, ...)
```

Arguments

x	An object to be converted into a tidy tibble::tibble().
...	Additional arguments to tidying method.

Value

a tibble

translate — *Resolve a Model Specification for a Computational Engine*

Description

`translate()` will translate a model specification into a code object that is specific to a particular engine (e.g. R package). It translates generic parameters to their counterparts.

Usage

```
translate(x, ...)

## Default S3 method:
translate(x, engine = x$engine, ...)
```

Arguments

`x`	A model specification.
`...`	Not currently used.
`engine`	The computational engine for the model (see `?set_engine`).

Details

`translate()` produces a *template* call that lacks the specific argument values (such as `data`, etc). These are filled in once `fit()` is called with the specifics of the data for the model. The call may also include `tune()` arguments if these are in the specification. To handle the `tune()` arguments, you need to use the tune package. For more information see `https://www.tidymodels.org/start/tuning/`

It does contain the resolved argument names that are specific to the model fitting function/engine.

This function can be useful when you need to understand how parsnip goes from a generic model specific to a model fitting function.

Note: this function is used internally and users should only use it to understand what the underlying syntax would be. It should not be used to modify the model specification.

Examples

```
lm_spec <- linear_reg(penalty = 0.01)

# `penalty` is tranlsated to `lambda`
translate(lm_spec, engine = "glmnet")

# `penalty` not applicable for this model.
translate(lm_spec, engine = "lm")

# `penalty` is tranlsated to `reg_param`
translate(lm_spec, engine = "spark")
```

```
# with a placeholder for an unknown argument value:
translate(linear_reg(penalty = tune(), mixture = tune()), engine = "glmnet")
```

---

update.bag_mars *Updating a model specification*

Description

If parameters of a model specification need to be modified, `update()` can be used in lieu of recreating the object from scratch.

Usage

```
## S3 method for class 'bag_mars'
update(
  object,
  parameters = NULL,
  num_terms = NULL,
  prod_degree = NULL,
  prune_method = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'bag_mlp'
update(
  object,
  parameters = NULL,
  hidden_units = NULL,
  penalty = NULL,
  epochs = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'bag_tree'
update(
  object,
  parameters = NULL,
  cost_complexity = NULL,
  tree_depth = NULL,
  min_n = NULL,
  class_cost = NULL,
  fresh = FALSE,
  ...
)
```

```
## S3 method for class 'bart'
update(
  object,
  parameters = NULL,
  trees = NULL,
  prior_terminal_node_coef = NULL,
  prior_terminal_node_expo = NULL,
  prior_outcome_range = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'boost_tree'
update(
  object,
  parameters = NULL,
  mtry = NULL,
  trees = NULL,
  min_n = NULL,
  tree_depth = NULL,
  learn_rate = NULL,
  loss_reduction = NULL,
  sample_size = NULL,
  stop_iter = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'C5_rules'
update(
  object,
  parameters = NULL,
  trees = NULL,
  min_n = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'cubist_rules'
update(
  object,
  parameters = NULL,
  committees = NULL,
  neighbors = NULL,
  max_rules = NULL,
  fresh = FALSE,
  ...
```

```
)

## S3 method for class 'decision_tree'
update(
  object,
  parameters = NULL,
  cost_complexity = NULL,
  tree_depth = NULL,
  min_n = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'discrim_flexible'
update(
  object,
  num_terms = NULL,
  prod_degree = NULL,
  prune_method = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'discrim_linear'
update(
  object,
  penalty = NULL,
  regularization_method = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'discrim_quad'
update(object, regularization_method = NULL, fresh = FALSE, ...)

## S3 method for class 'discrim_regularized'
update(
  object,
  frac_common_cov = NULL,
  frac_identity = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'gen_additive_mod'
update(
  object,
  select_features = NULL,
  adjust_deg_free = NULL,
  parameters = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'linear_reg'
update(
  object,
  parameters = NULL,
  penalty = NULL,
  mixture = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'logistic_reg'
update(
  object,
  parameters = NULL,
  penalty = NULL,
  mixture = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'mars'
update(
  object,
  parameters = NULL,
  num_terms = NULL,
  prod_degree = NULL,
  prune_method = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'mlp'
update(
  object,
  parameters = NULL,
  hidden_units = NULL,
  penalty = NULL,
  dropout = NULL,
  epochs = NULL,
  activation = NULL,
  learn_rate = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'multinom_reg'
update(
  object,
  parameters = NULL,
  penalty = NULL,
  mixture = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'naive_Bayes'
update(object, smoothness = NULL, Laplace = NULL, fresh = FALSE, ...)

## S3 method for class 'nearest_neighbor'
update(
  object,
  parameters = NULL,
  neighbors = NULL,
  weight_func = NULL,
  dist_power = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'pls'
update(
  object,
  parameters = NULL,
  predictor_prop = NULL,
  num_comp = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'poisson_reg'
update(
  object,
  parameters = NULL,
  penalty = NULL,
  mixture = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'proportional_hazards'
```

```
update(
  object,
  parameters = NULL,
  penalty = NULL,
  mixture = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'rand_forest'
update(
  object,
  parameters = NULL,
  mtry = NULL,
  trees = NULL,
  min_n = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'rule_fit'
update(
  object,
  parameters = NULL,
  mtry = NULL,
  trees = NULL,
  min_n = NULL,
  tree_depth = NULL,
  learn_rate = NULL,
  loss_reduction = NULL,
  sample_size = NULL,
  penalty = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'surv_reg'
update(object, parameters = NULL, dist = NULL, fresh = FALSE, ...)

## S3 method for class 'survival_reg'
update(object, parameters = NULL, dist = NULL, fresh = FALSE, ...)

## S3 method for class 'svm_linear'
update(
  object,
  parameters = NULL,
  cost = NULL,
  margin = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'svm_poly'
update(
  object,
  parameters = NULL,
  cost = NULL,
  degree = NULL,
  scale_factor = NULL,
  margin = NULL,
  fresh = FALSE,
  ...
)

## S3 method for class 'svm_rbf'
update(
  object,
  parameters = NULL,
  cost = NULL,
  rbf_sigma = NULL,
  margin = NULL,
  fresh = FALSE,
  ...
)
```

Arguments

`object`	A model specification.
`parameters`	A 1-row tibble or named list with *main* parameters to update. Use **either** `parameters` **or** the main arguments directly when updating. If the main arguments are used, these will supersede the values in `parameters`. Also, using engine arguments in this object will result in an error.
`num_terms`	The number of features that will be retained in the final model, including the intercept.
`prod_degree`	The highest possible interaction degree.
`prune_method`	The pruning method.
`fresh`	A logical for whether the arguments should be modified in-place or replaced wholesale.
`...`	Not used for `update()`.
`hidden_units`	An integer for the number of units in the hidden model.
`penalty`	An non-negative number representing the amount of regularization used by some of the engines.
`epochs`	An integer for the number of training iterations.

`cost_complexity`	A positive number for the the cost/complexity parameter (a.k.a. `Cp`) used by CART models (specific engines only).
`tree_depth`	An integer for maximum depth of the tree.
`min_n`	An integer for the minimum number of data points in a node that are required for the node to be split further.
`class_cost`	A non-negative scalar for a class cost (where a cost of 1 means no extra cost). This is useful for when the first level of the outcome factor is the minority class. If this is not the case, values between zero and one can be used to bias to the second level of the factor.
`trees`	An integer for the number of trees contained in the ensemble.
`prior_terminal_node_coef`	A coefficient for the prior probability that a node is a terminal node.
`prior_terminal_node_expo`	An exponent in the prior probability that a node is a terminal node.
`prior_outcome_range`	A positive value that defines the width of a prior that the predicted outcome is within a certain range. For regression it is related to the observed range of the data; the prior is the number of standard deviations of a Gaussian distribution defined by the observed range of the data. For classification, it is defined as the range of +/-3 (assumed to be on the logit scale). The default value is 2.
`mtry`	A number for the number (or proportion) of predictors that will be randomly sampled at each split when creating the tree models (specific engines only).
`learn_rate`	A number for the rate at which the boosting algorithm adapts from iteration-to-iteration (specific engines only). This is sometimes referred to as the shrinkage parameter.
`loss_reduction`	A number for the reduction in the loss function required to split further (specific engines only).
`sample_size`	A number for the number (or proportion) of data that is exposed to the fitting routine. For `xgboost`, the sampling is done at each iteration while `C5.0` samples once during training.
`stop_iter`	The number of iterations without improvement before stopping (specific engines only).
`committees`	A non-negative integer (no greater than 100) for the number of members of the ensemble.
`neighbors`	An integer between zero and nine for the number of training set instances that are used to adjust the model-based prediction.
`max_rules`	The largest number of rules.
`regularization_method`	A character string for the type of regularized estimation. Possible values are: `"diagonal"`, `"min_distance"`, `"shrink_cov"`, and `"shrink_mean"` (`sparsediscrim` engine only).
`frac_common_cov`, `frac_identity`	Numeric values between zero and one.

`select_features`	`TRUE` or `FALSE.` If `TRUE`, the model has the ability to eliminate a predictor (via penalization). Increasing `adjust_deg_free` will increase the likelihood of removing predictors.
`adjust_deg_free`	If `select_features = TRUE`, then acts as a multiplier for smoothness. Increase this beyond 1 to produce smoother models.
`mixture`	A number between zero and one (inclusive) denoting the proportion of L1 regularization (i.e. lasso) in the model. • `mixture = 1` specifies a pure lasso model, • `mixture = 0` specifies a ridge regression model, and • `0 < mixture < 1` specifies an elastic net model, interpolating lasso and ridge. Available for specific engines only.
`dropout`	A number between 0 (inclusive) and 1 denoting the proportion of model parameters randomly set to zero during model training.
`activation`	A single character string denoting the type of relationship between the original predictors and the hidden unit layer. The activation function between the hidden and output layers is automatically set to either "linear" or "softmax" depending on the type of outcome. Possible values depend on the engine being used.
`smoothness`	An non-negative number representing the the relative smoothness of the class boundary. Smaller examples result in model flexible boundaries and larger values generate class boundaries that are less adaptable
`Laplace`	A non-negative value for the Laplace correction to smoothing low-frequency counts.
`weight_func`	A *single* character for the type of kernel function used to weight distances between samples. Valid choices are: `"rectangular"`, `"triangular"`, `"epanechnikov"`, `"biweight"`, `"triweight"`, `"cos"`, `"inv"`, `"gaussian"`, `"rank"`, or `"optimal"`.
`dist_power`	A single number for the parameter used in calculating Minkowski distance.
`predictor_prop`	The maximum proportion of original predictors that can have *non-zero* coefficients for each PLS component (via regularization). This value is used for all PLS components for X.
`num_comp`	The number of PLS components to retain.
`dist`	A character string for the probability distribution of the outcome. The default is "weibull".
`cost`	A positive number for the cost of predicting a sample within or on the wrong side of the margin
`margin`	A positive number for the epsilon in the SVM insensitive loss function (regression only)
`degree`	A positive number for polynomial degree.
`scale_factor`	A positive number for the polynomial scaling factor.
`rbf_sigma`	A positive number for radial basis function.

Value

An updated model specification.

Examples

```
# ------------------------------------------------------------------------------

model <- C5_rules(trees = 10, min_n = 2)
model
update(model, trees = 1)
update(model, trees = 1, fresh = TRUE)



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

model <- cubist_rules(committees = 10, neighbors = 2)
model
update(model, committees = 1)
update(model, committees = 1, fresh = TRUE)


model <- pls(predictor_prop =  0.1)
model
update(model, predictor_prop = 1)
update(model, predictor_prop = 1, fresh = TRUE)


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

model <- rule_fit(trees = 10, min_n = 2)
model
update(model, trees = 1)
update(model, trees = 1, fresh = TRUE)


model <- boost_tree(mtry = 10, min_n = 3)
model
update(model, mtry = 1)
update(model, mtry = 1, fresh = TRUE)

param_values <- tibble::tibble(mtry = 10, tree_depth = 5)

model %>% update(param_values)
model %>% update(param_values, mtry = 3)

param_values$verbose <- 0
# Fails due to engine argument
# model %>% update(param_values)

model <- linear_reg(penalty = 10, mixture = 0.1)
```

```
model
update(model, penalty = 1)
update(model, penalty = 1, fresh = TRUE)
```

Index