Package ‘parsnip’

January 4, 2020

Version 0.0.5
Title A Common API to Modeling and Analysis Functions
Description A common interface is provided to allow users to specify a model without hav-
ing to remember the different argument names across different functions or computa-
tional engines (e.g. ‘R’, ‘Spark’, ‘Stan’, etc).
Maintainer Max Kuhn <max@rstudio.com>
BugReports https://github.com/tidymodels/parsnip/issues
License GPL-2
Encoding UTF-8
LazyData true
ByteCompile true
VignetteBuilder knitr
Depends R (>= 2.10)
Imports dplyr (>= 0.8.0.1),
   rlang (>= 0.3.1),
   purrr,
   utils,
   tibble (>= 2.1.1),
   generics,
   glue,
   magrittr,
   stats,
   tidyr,
   globals,
   prettyunits,
   vctrs (>= 0.2.0)
Roxygen Roxygen list(markdown = TRUE)
RoxygenNote 7.0.2.9000
Suggests testthat,
   knitr,
   rmarkdown,
   survival,
   keras,
   xgboost,
R topics documented:

covr,
C50,
sparklyr ($>= 1.0.0$),
earth,
kernlab,
kknn,
randomForest,
ranger,
rpart,
MASS,
nlme,
modeldata

R topics documented:

add_rowindex .................................................. 3
boost_tree ..................................................... 3
control_parsnip ............................................... 6
decision_tree .................................................. 7
descriptors .................................................... 9
fit.model_spec ................................................. 11
linear_reg ..................................................... 12
logistic_reg ................................................... 15
mars ............................................................... 17
mlp ............................................................... 19
model_fit ........................................................ 22
model_spec ..................................................... 23
multinom_reg ................................................... 25
multi_predict .................................................. 27
nearest_neighbor ............................................... 28
nullmodel ...................................................... 29
null_model ...................................................... 31
predict.model_fit ............................................. 32
rand_forest ..................................................... 34
set_args ....................................................... 36
set_engine ...................................................... 37
surv_reg ....................................................... 37
svm_poly ....................................................... 39
svm_rbf ....................................................... 41
tidy.model_fit ................................................ 43
translate ........................................................ 43
varying .......................................................... 44
varying_args.model_spec ..................................... 44

Index 46
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()

boost_tree

General Interface for Boosted Trees

Description
boost_tree() is a way to generate a specification of a model before fitting and allows the model to be created using different packages in R or via Spark. The main arguments for the model are:

- mtry: The number of predictors that will be randomly sampled at each split when creating the tree models.
- trees: The number of trees contained in the ensemble.
- min_n: The minimum number of data points in a node that are required for the node to be split further.
- tree_depth: The maximum depth of the tree (i.e. number of splits).
- learn_rate: The rate at which the boosting algorithm adapts from iteration-to-iteration.
- loss_reduction: The reduction in the loss function required to split further.
- sample_size: The amount of data exposed to the fitting routine.

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using the set_engine() function. If left to their defaults here (NULL), the values are taken from the underlying model functions. If parameters need to be modified, update() can be used in lieu of recreating the object from scratch.
Usage

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

## 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,
  fresh = FALSE,
  ...
)

Arguments

mode A single character string for the type of model. Possible values for this
model are "unknown", "regression", or "classification".
mtry An number for the number (or proportion) of predictors that will be
randomly sampled at each split when creating the tree models (xgboost
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 are
required for the node to be split further.
tree_depth An integer for the maximum deopth of the tree (i.e. number of splits)
( xgboost only).
learn_rate A number for the rate at which the boosting algorithm adapts from
iteration-to-iteration (xgboost only).
loss_reduction A number for the reduction in the loss function required to split further
(xgboost only).
sample_size An number for the number (or proportion) of data that is exposed to the
fitting routine. For xgboost, the sampling is done at at each iteration
while \$5.0\$ samples once during traning.
object A boosted tree model specification.
parameters A 1-row tibble or named list with main parameters to update. If the indi-
vidual arguments are used, these will supersede the values in parameters.
Also, using engine arguments in this object will result in an error.
boost_tree

fresh A logical for whether the arguments should be modified in-place of or replaced wholesale.

... Not used for update().

Details

The data given to the function are not saved and are only used to determine the mode of the model. For boost_tree(), the possible modes are "regression" and "classification".

The model can be created using the fit() function using the following engines:

- R: "xgboost" (the default), "C5.0"
- Spark: "spark"

Value

An updated model specification.

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:

**xgboost** classification

parsnip::xgb_train(x = missing_arg(), y = missing_arg(), nthread = 1, verbose = 0)

**xgboost** regression

parsnip::xgb_train(x = missing_arg(), y = missing_arg(), nthread = 1, verbose = 0)

**C5.0** classification

parsnip::C5.0_train(x = missing_arg(), y = missing_arg(), weights = missing_arg())

**spark** classification

sparklyr::ml_gradient_boosted_trees(x = missing_arg(), formula = missing_arg(), type = "classification", seed = sample.int(10^5, 1))

**spark** regression

sparklyr::ml_gradient_boosted_trees(x = missing_arg(), formula = missing_arg(), type = "regression", seed = sample.int(10^5, 1))

Note

For models created using the spark engine, there are several differences to consider. First, only the formula interface to via fit() is available; using fit.xy() will generate an error. Second, the predictions will always be in a spark table format. The names will be the same as documented but without the dots. Third, there is no equivalent to factor columns in spark tables so class predictions are returned as character columns. Fourth, to retain the model object for a new R session (via save()), the model$fit element of the parsnip object should be serialized via ml_save(object$fit) and separately saved to disk. In a new session, the object can be reloaded and reattached to the parsnip object.
See Also

[fit()], set_engine()

Examples

boost_tree(mode = "classification", trees = 20)
# Parameters can be represented by a placeholder:
boost_tree(mode = "regression", mtry = varying())
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)

control_parsnip

Control the fit function

Description

Options can be passed to the fit() function that control the output and computations

Usage

control_parsnip(verbosity = 1L, catch = FALSE)

fit_control(verbosity = 1L, catch = FALSE)

Arguments

verbosity An integer where a value of zero indicates that no messages or output
should be shown when packages are loaded or when the model is fit. A
value of 1 means that package loading is quiet but model fits can produce
output to the screen (depending on if they contain their own verbose-
type argument). A value of 2 or more indicates that any output should
be seen.

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".

Details

fit_control() is deprecated in favor of control_parsnip().
**Value**

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

---

### decision_tree

**General Interface for Decision Tree Models**

**Description**

decision_tree() is a way to generate a specification of a model before fitting and allows the model to be created using different packages in R or via Spark. The main arguments for the model are:

- **cost/complexity**: The cost/complexity parameter (a.k.a. Cp) used by CART models (rpart only).
- **tree depth**: The maximum depth of a tree (rpart and spark only).
- **min_n**: The minimum number of data points in a node that are required for the node to be split further.

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using set_engine(). If left to their defaults here (NULL), the values are taken from the underlying model functions. If parameters need to be modified, update() can be used in lieu of recreating the object from scratch.

**Usage**

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

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

**Arguments**

- **mode**
  A single character string for the type of model. Possible values for this model are "unknown", "regression", or "classification".

- **cost_complexity**
  A positive number for the the cost/complexity parameter (a.k.a. Cp) used by CART models (rpart only).
The model can be created using the `fit()` function using the following **engines**:

- **R**: "rpart" (the default) or "C5.0" (classification only)
- **Spark**: "spark"

Note that, for rpart models, but `cost_complexity` and `tree_depth` can be both be specified but the package will give precedence to `cost_complexity`. Also, `tree_depth` values greater than 30 rpart will give nonsense results on 32-bit machines.

### 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:

**rpart** classification

```r
rpart::rpart(formula = missing_arg(), data = missing_arg(), weights = missing_arg())
```

**rpart** regression

```r
rpart::rpart(formula = missing_arg(), data = missing_arg(), weights = missing_arg())
```

**C5.0** classification

```r
parsnip::C5.0_train(x = missing_arg(), y = missing_arg(), weights = missing_arg(), trials = 1)
```

**spark** classification

```r
sparklyr::ml_decision_tree_classifier(x = missing_arg(), formula = missing_arg(), seed = sample.int(10^5, 1))
```

**spark** regression

```r
sparklyr::ml_decision_tree_classifier(x = missing_arg(), formula = missing_arg(), seed = sample.int(10^5, 1))
```
Note

For models created using the spark engine, there are several differences to consider. First, only the formula interface to via `fit()` is available; using `fit_xy()` will generate an error. Second, the predictions will always be in a spark table format. The names will be the same as documented but without the dots. Third, there is no equivalent to factor columns in spark tables so class predictions are returned as character columns. Fourth, to retain the model object for a new R session (via `save()`), the `model$fit` element of the `parsnip` object should be serialized via `ml_save(object$fit)` and separately saved to disk. In a new session, the object can be reloaded and reattached to the `parsnip` object.

See Also

`[fit()]`

Examples

decision_tree(mode = "classification", tree_depth = 5)
# Parameters can be represented by a placeholder:
decision_tree(mode = "regression", cost_complexity = varying())
model <- decision_tree(cost_complexity = 10, min_n = 3)
model
update(model, cost_complexity = 1)
update(model, cost_complexity = 1, fresh = TRUE)

<table>
<thead>
<tr>
<th>descriptors</th>
<th>Data Set Characteristics Available when Fitting Models</th>
</tr>
</thead>
</table>

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()
.lvlsl()
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 are 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 `Sepal.Width ~ .` with the `iris` data, the values would be

```
.preds() = 4  (the 4 columns in `iris`)
.cols() = 5   (3 numeric columns + 2 from Species dummy variables)
.obs() = 150
.lvls() = NA  (no factor outcome)
.facts() = 1  (the Species predictor)
.y() = <vector>  (Sepal.Width as a vector)
.x() = <data.frame> (The other 4 columns as a data frame)
.dat() = <data.frame> (The full data set)
```

If the formula `Species ~ .` were used:

```
.preds() = 4  (the 4 numeric columns in `iris`)
.cols() = 4   (same)
.obs() = 150
.lvls() = c(setosa = 50, versicolor = 50, virginica = 50)
.facts() = 0
.y() = <vector>  (Species as a vector)
.x() = <data.frame> (The other 4 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:

```r
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.
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, control = control_parsnip(), ...)

## S3 method for class 'model_spec'
fit_xy(object, x, y, 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.
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 or data frame of predictors.
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, checks them for validity, then fits 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() were 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.
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

```r
# 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
```

linear_reg

---

**General Interface for Linear Regression Models**

Description

`linear_reg()` is a way to generate a *specification* of a model before fitting and allows the model to be created using different packages in R, Stan, keras, or via Spark. The main arguments for the model are:
- **penalty**: The total amount of regularization in the model. Note that this must be zero for some engines.
- **mixture**: The proportion of L1 regularization in the model. Note that this will be ignored for some engines.

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using `set_engine()`. If left to their defaults here (NULL), the values are taken from the underlying model functions. If parameters need to be modified, `update()` can be used in lieu of recreating the object from scratch.

**Usage**

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

**Arguments**

- `mode` A single character string for the type of model. The only possible value for this model is "regression".
- `penalty` An non-negative number representing the total amount of regularization (glmnet, keras, and spark 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) that represents the proportion of regularization that is used for the L2 penalty (i.e. weight decay, or ridge regression) versus L1 (the lasso) (glmnet and spark only).
- `object` A linear regression model specification.
- `parameters` A 1-row tibble or named list with main parameters to update. If the individual arguments are used, these will supersede the values in `parameters`. Also, using engine arguments in this object will result in an error.
- `fresh` A logical for whether the arguments should be modified in-place of or replaced wholesale.

**Details**

The data given to the function are not saved and are only used to determine the `mode` of the model. For `linear_reg()`, the mode will always be "regression".

The model can be created using the `fit()` function using the following engines:

- R: "lm" (the default) or "glmnet"
- Stan: "stan"
- Spark: "spark"
- keras: "keras"
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:

**lm**

stats::lm(formula = missing_arg(), data = missing_arg(), weights = missing_arg())

**glmnet**

glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(), family = "gaussian")

**stan**

rstanarm::stan_glm(formula = missing_arg(), data = missing_arg(), weights = missing_arg(), family = stats::gaussian)

**spark**

sparklyr::ml_linear_regression(x = missing_arg(), formula = missing_arg(), weight_col = missing_arg())

**keras**

parsnip::keras_mlp(x = missing_arg(), y = missing_arg(), hidden_units = 1, act = "linear")

For glmnet models, the full regularization path is always fit regardless of the value given to penalty. Also, there is the option to pass multiple values (or no values) to the penalty argument. When using the predict() method in these cases, the return value depends on the value of penalty. When using predict(), only a single value of the penalty can be used. When predicting on multiple penalties, the multi_predict() function can be used. It returns a tibble with a list column called .pred that contains a tibble with all of the penalty results.

For prediction, the stan engine can compute posterior intervals analogous to confidence and prediction intervals. In these instances, the units are the original outcome and when std.error = TRUE, the standard deviation of the posterior distribution (or posterior predictive distribution as appropriate) is returned.

**Note**

For models created using the spark engine, there are several differences to consider. First, only the formula interface to via fit() is available; using fit.xy() will generate an error. Second, the predictions will always be in a spark table format. The names will be the same as documented but without the dots. Third, there is no equivalent to factor columns in spark tables so class predictions are returned as character columns. Fourth, to retain the model object for a new R session (via save()), the model$fit element of the parsnip object should be serialized via ml_save(object$fit) and separately saved to disk. In a new session, the object can be reloaded and reattached to the parsnip object.
logistic_reg

See Also
[fit(), set_engine()]

Examples
linear_reg()
# Parameters can be represented by a placeholder:
linear_reg(penalty = varying())
model <- linear_reg(penalty = 10, mixture = 0.1)
model
update(model, penalty = 1)
update(model, penalty = 1, fresh = TRUE)

#### logistic_reg

**General Interface for Logistic Regression Models**

**Description**
logistic_reg() is a way to generate a specification of a model before fitting and allows the model to be created using different packages in R, Stan, keras, or via Spark. The main arguments for the model are:

- **penalty**: The total amount of regularization in the model. Note that this must be zero for some engines.
- **mixture**: The proportion of L1 regularization in the model. Note that this will be ignored for some engines.

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using set_engine(). If left to their defaults here (NULL), the values are taken from the underlying model functions. If parameters need to be modified, update() can be used in lieu of recreating the object from scratch.

**Usage**

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

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

**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 total amount of regularization (glmnet, keras, and spark 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) that represents the proportion of regularization that is used for the L2 penalty (i.e. weight decay, or ridge regression) versus L1 (the lasso) (glmnet and spark only).

object
A logistic regression model specification.

parameters
A 1-row tibble or named list with *main* parameters to update. If the individual arguments are used, these will supersede the values in parameters. Also, using engine arguments in this object will result in an error.

fresh
A logical for whether the arguments should be modified in-place of or replaced wholesale.

... Not used for update().

Details
For logistic_reg(), the mode will always be "classification".
The model can be created using the fit() function using the following engines:

- R: "glm" (the default) or "glmnet"
- Stan: "stan"
- Spark: "spark"
- keras: "keras"

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:

**glm**

```r
stats::glm(formula = missing_arg(), data = missing_arg(), weights = missing_arg(),
            family = stats::binomial)
```

**glmnet**

```r
glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
                family = "binomial")
```

**stan**

```r
rstanarm::stan_glm(formula = missing_arg(), data = missing_arg(),
                   weights = missing_arg(), family = stats::binomial)
```

**spark**

```r
sparklyr::ml_logistic_regression(x = missing_arg(), formula = missing_arg(),
                                 weight_col = missing_arg(), family = "binomial")
```

**keras**
For glmnet models, the full regularization path is always fit regardless of the value given to penalty. Also, there is the option to pass multiple values (or no values) to the penalty argument. When using the predict() method in these cases, the return value depends on the value of penalty. When using predict(), only a single value of the penalty can be used. When predicting on multiple penalties, the multi_predict() function can be used. It returns a tibble with a list column called .pred that contains a tibble with all of the penalty results.

For prediction, the stan engine can compute posterior intervals analogous to confidence and prediction intervals. In these instances, the units are the original outcome and when std_error = TRUE, the standard deviation of the posterior distribution (or posterior predictive distribution as appropriate) is returned. For glm, the standard error is in logit units while the intervals are in probability units.

**Note**

For models created using the spark engine, there are several differences to consider. First, only the formula interface to via fit() is available; using fit_xy() will generate an error. Second, the predictions will always be in a spark table format. The names will be the same as documented but without the dots. Third, there is no equivalent to factor columns in spark tables so class predictions are returned as character columns. Fourth, to retain the model object for a new R session (via save()), the model$fit element of the parsnip object should be serialized via ml_save(object$fit) and separately saved to disk. In a new session, the object can be reloaded and reattached to the parsnip object.

**See Also**

[fit()](#)

**Examples**

logistic_reg()

# Parameters can be represented by a placeholder:
logistic_reg(penalty = varying())

model <- logistic_reg(penalty = 10, mixture = 0.1)
model
update(model, penalty = 1)
update(model, penalty = 1, fresh = TRUE)

---

mars is a way to generate a specification of a model before fitting and allows the model to be created using R. The main arguments for the model are:

- **num_terms**: The number of features that will be retained in the final model.
- **prod_degree**: The highest possible degree of interaction between features. A value of 1 indicates and additive model while a value of 2 allows, but does not guarantee, two-way interactions between features.
• prune_method: The type of pruning. Possible values are listed in ?earth.

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using set_engine(). If left to their defaults here (NULL), the values are taken from the underlying model functions. If parameters need to be modified, update() can be used in lieu of recreating the object from scratch.

Usage

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

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

Arguments

mode A single character string for the type of model. 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.

object A MARS model specification.

parameters A 1-row tibble or named list with main parameters to update. If the individual arguments are used, these will supersede the values in parameters. Also, using engine arguments in this object will result in an error.

fresh A logical for whether the arguments should be modified in-place of or replaced wholesale.

... Not used for update().

Details

The model can be created using the fit() function using the following engines:

• R: "earth" (the default)
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:

**earth** classification

```r
earth::earth(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
glm = list(family = stats::binomial), keepxy = TRUE)
```

**earth** regression

```r
earth::earth(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
keepxy = TRUE)
```

Note that, when the model is fit, the **earth** package only has its namespace loaded. However, if `multi_predict` is used, the package is attached.

See Also

`[fit()]`

Examples

```r
mars(mode = "regression", num_terms = 5)
model <- mars(num_terms = 10, prune_method = "none")
model
update(model, num_terms = 1)
update(model, num_terms = 1, fresh = TRUE)
```

---

**mlp**

*General Interface for Single Layer Neural Network*

Description

`mlp()`, for multilayer perceptron, is a way to generate a specification of a model before fitting and allows the model to be created using different packages in R or via keras. The main arguments for the model are:

- `hidden_units`: The number of units in the hidden layer (default: 5).
- `penalty`: The amount of L2 regularization (aka weight decay, default is zero).
- `dropout`: The proportion of parameters randomly dropped out of the model (keras only, default is zero).
- `epochs`: The number of training iterations (default: 20).
- `activation`: The type of function that connects the hidden layer and the input variables (keras only, default is softmax).

If parameters need to be modified, this function can be used in lieu of recreating the object from scratch.
Usage

```r
mlp(
  mode = "unknown",
  hidden_units = NULL,
  penalty = NULL,
  dropout = NULL,
  epochs = NULL,
  activation = NULL
)
```

## S3 method for class 'mlp'

```r
update(
  object,
  parameters = NULL,
  hidden_units = NULL,
  penalty = NULL,
  dropout = NULL,
  epochs = NULL,
  activation = NULL,
  fresh = FALSE,
  ...
)
```

Arguments

- **mode**
  A single character string for the type of model. 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.

- **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 are: "linear", "softmax", "relu", and "elu"

- **object**
  A random forest model specification.

- **parameters**
  A 1-row tibble or named list with main parameters to update. If the individual arguments are used, these will supersede the values in `parameters`. Also, using engine arguments in this object will result in an error.

- **fresh**
  A logical for whether the arguments should be modified in-place of or replaced wholesale.

- **...**
  Not used for `update()`.

Details

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using `set_engine()`. If left to their defaults here (see above), the values are taken from the underlying model functions. One exception is
hidden_units when nnet::nnet is used; that function’s size argument has no default so a value of 5 units will be used. Also, unless otherwise specified, the linout argument to nnet::nnet() will be set to TRUE when a regression model is created. If parameters need to be modified, update() can be used in lieu of recreating the object from scratch.

The model can be created using the fit() function using the following engines:

- R: "nnet" (the default)
- keras: "keras"

An error is thrown if both penalty and dropout are specified for keras models.

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:

**keras** classification

parsnip::keras_mlp(x = missing_arg(), y = missing_arg())

**keras** regression

parsnip::keras_mlp(x = missing_arg(), y = missing_arg())

**nnet** classification

nnet::nnet(formula = missing_arg(), data = missing_arg(), weights = missing_arg(), size = 5, trace = FALSE, linout = FALSE)

**nnet** regression

nnet::nnet(formula = missing_arg(), data = missing_arg(), weights = missing_arg(), size = 5, trace = FALSE, linout = TRUE)

See Also

[fit()]

Examples

mlp(mode = "classification", penalty = 0.01)
# Parameters can be represented by a placeholder:
mlp(mode = "regression", hidden_units = varying())
model <- mlp(hidden_units = 10, dropout = 0.30)
model
update(model, hidden_units = 2)
update(model, hidden_units = 2, fresh = TRUE)
Model Fit Object Information

Description

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

Details

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 to seeing the data and applying a model.

Examples

```r
# 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_spec**  

**Description**

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

**Details**

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 in 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 varying(). 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 varying().

- **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

```r
rand_forest(mtry = ncol(iris) - 1)
```

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

```r
> rand_forest(mtry = ncol(iris) - 1)
Random Forest Model Specification (unknown)
Main Arguments:
  mtry = ncol(iris) - 1
```
The model functions save the argument *expressions* and their associated environments (a.k.a. a quosure) to be evaluated later when either `fit()` or `fit_xy()` 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

```r
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

```r
rand_forest(mtry = ncol(!!iris) - 1)
```

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

```r
> rand_forest(mtry = ncol(!!iris) - 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:

```r
> mtry_val <- ncol(iris) - 1
> mtry_val
[1] 4
> rand_forest(mtry = !!mtry_val)
Random Forest Model Specification (unknown)
Main Arguments:
mtry = 4
```

More information on quosures and quasiquotation can be found at [https://tidyeval.tidyverse.org](https://tidyeval.tidyverse.org).
multinom_reg

General Interface for Multinomial Regression Models

Description

multinom_reg() is a way to generate a specification of a model before fitting and allows the model to be created using different packages in R, keras, or Spark. The main arguments for the model are:

- **penalty**: The total amount of regularization in the model. Note that this must be zero for some engines.
- **mixture**: The proportion of L1 regularization in the model. Note that this will be ignored for some engines.

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using `set_engine()`. If left to their defaults here (NULL), the values are taken from the underlying model functions. If parameters need to be modified, `update()` can be used in lieu of recreating the model functions. If parameters need to be modified, `update()` can be used in lieu of recreating the model functions. If parameters need to be modified, `update()` can be used in lieu of recreating the model functions.

Usage

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

## S3 method for class 'multinom_reg'

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

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 total amount of regularization (glmnet, keras, and spark 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) that represents the proportion of regularization that is used for the L2 penalty (i.e. weight decay, or ridge regression) versus L1 (the lasso) (glmnet only).
- **object**: A multinomial regression model specification.
- **parameters**: A 1-row tibble or named list with main parameters to update. If the individual arguments are used, these will supersede the values in `parameters`. Also, using engine arguments in this object will result in an error.
- **fresh**: A logical for whether the arguments should be modified in-place of or replaced wholesale.
- **...**: Not used for `update()`.
Details

For `multinom_reg()`, the mode will always be "classification".
The model can be created using the `fit()` function using the following engines:

- **R**: "glmnet" (the default), "nnet"
- **Stan**: "stan"
- **keras**: "keras"

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:

**glmnet**

```r
glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
               family = "multinomial")
```

**nnet**

```r
nnet::multinom(formula = missing_arg(), data = missing_arg(),
               weights = missing_arg(), trace = FALSE)
```

**spark**

```r
sparklyr::ml_logistic_regression(x = missing_arg(), formula = missing_arg(),
                                 weight_col = missing_arg(), family = "multinomial")
```

**keras**

```r
parsnip::keras_mlp(x = missing_arg(), y = missing_arg(), hidden_units = 1,
                   act = "linear")
```

For `glmnet` models, the full regularization path is always fit regardless of the value given to `penalty`. Also, there is the option to pass multiple values (or no values) to the `penalty` argument. When using the `predict()` method in these cases, the return value depends on the value of `penalty`. When using `predict()`, only a single value of the penalty can be used. When predicting on multiple penalties, the `multi_predict()` function can be used. It returns a tibble with a list column called `.pred` that contains a tibble with all of the penalty results.

**Note**

For models created using the spark engine, there are several differences to consider. First, only the formula interface to `fit()` is available; using `fit_xy()` will generate an error. Second, the predictions will always be in a spark table format. The names will be the same as documented but without the dots. Third, there is no equivalent to factor columns in spark tables so class predictions are returned as character columns. Fourth, to retain the model object for a new R session (via `save()`), the model$fit element of the `parsnip` object should be serialized via `ml_save(object$fit)` and separately saved to disk. In a new session, the object can be reloaded and reattached to the `parsnip` object.
multi_predict

See Also

[fit()]

Examples

multinom_reg()

# Parameters can be represented by a placeholder:
multinom_reg(penalty = varying())
model <- multinom_reg(penalty = 10, mixture = 0.1)
model
update(model, penalty = 1)
update(model, penalty = 1, fresh = TRUE)

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 "earth"
multi_predict(object, new_data, type = NULL, num_terms = NULL, ...)

## S3 method for class "multnet"
multi_predict(object, new_data, type = NULL, penalty = NULL, ...)

## S3 method for class "train.kknn"
multi_predict(object, new_data, type = NULL, neighbors = NULL, ...)
Arguments

- **object**: A `model_fit` object.
- **...**: 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**: An numeric vector of penalty values.
- **num_terms**: An integer vector for the number of MARS terms to retain.
- **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).

Description

`nearest_neighbor()` is a way to generate a *specification* of a model before fitting and allows the model to be created using different packages in R. The main arguments for the model are:

- **neighbors**: The number of neighbors considered at each prediction.
- **weight_func**: The type of kernel function that weights the distances between samples.
- **dist_power**: The parameter used when calculating the Minkowski distance. This corresponds to the Manhattan distance with `dist_power = 1` and the Euclidean distance with `dist_power = 2`.

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using `set_engine()`. If left to their defaults here (`NULL`), the values are taken from the underlying model functions. If parameters need to be modified, `update()` can be used in lieu of recreating the object from scratch.

Usage

```r
nearest_neighbor(
  mode = "unknown",
  neighbors = NULL,
  weight_func = NULL,
  dist_power = NULL
)
```
nullmodel

Arguments

- **mode**: A single character string for the type of model. Possible values for this model are "unknown", "regression", or "classification".
- **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

The model can be created using the **fit()** function using the following **engines**:

- **R**: "kknn" (the default)

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:

- **kknn** (classification or regression)

```
kknn::train.kknn(formula = missing_arg(), data = missing_arg(),
                  ks = 5)
```

Note

For **kknn**, the underlying modeling function used is a restricted version of **train.kknn()** and not **kknn()**. It is set up in this way so that **parsnip** can utilize the underlying **predict.train.kknn** method to predict on new data. This also means that a single value of that function’s **kernel** argument (a.k.a **weight_func** here) can be supplied.

See Also

- **[fit()]**

Examples

```r
nearest_neighbor(neighbors = 11)
```

---

nullmodel: *Fit a simple, non-informative model*

Description

Fit a single mean or largest class model. **nullmodel()** is the underlying computational function for the **null_model()** specification.
nullmodel

Usage

nullmodel(x, ...)

## Default S3 method:
nullmodel(x = NULL, y, ...)

## S3 method for class 'nullmodel'
print(x, ...)

## S3 method for class 'nullmodel'
predict(object, new_data = NULL, type = NULL, ...)

Arguments

x      An optional matrix or data frame of predictors. These values are not used in the model fit
...
...   Optional arguments (not yet used)
y      A numeric vector (for regression) or factor (for classification) of outcomes
object An object of class nullmodel
new_data A matrix or data frame of predictors (only used to determine the number of predictions to return)
type   Either "raw" (for regression), "class" or "prob" (for classification)

Details

nullmodel() emulates other model building functions, but returns the simplest model possible given a training set: a single mean for numeric outcomes and the most prevalent class for factor outcomes. When class probabilities are requested, the percentage of the training set samples with the most prevalent class is returned.

Value

The output of nullmodel() is a list of class nullmodel with elements

call     the function call
value    the mean of y or the most prevalent class
levels   when y is a factor, a vector of levels. NULL otherwise
pct      when y is a factor, a data frame with a column for each class (NULL otherwise). The column for the most prevalent class has the proportion of the training samples with that class (the other columns are zero).
n        the number of elements in y

predict.nullmodel() returns a either a factor or numeric vector depending on the class of y. All predictions are always the same.

Examples

outcome <- factor(sample(letters[1:2],
                        size = 100,
                        prob = c(.1, .9),
                        ...
null_model

```r
replace = TRUE))
useless <- nullmodel(y = outcome)
useless
predict(useless, matrix(NA, nrow = 5))
```

---

**null_model**

*General Interface for null models*

**Description**

`null_model()` is a way to generate a specification of a model before fitting and allows the model to be created using R. It doesn’t have any main arguments.

**Usage**

```r
null_model(mode = "classification")
```

**Arguments**

- `mode` A single character string for the type of model. Possible values for this model are "unknown", "regression", or "classification".

**Details**

The model can be created using the `fit()` function using the following engines:

- **R**: "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:

- **parsnip** classification

  ```r
  nullmodel(x = missing_arg(), y = missing_arg())
  ```

- **parsnip** regression

  ```r
  nullmodel(x = missing_arg(), y = missing_arg())
  ```

**See Also**

- `fit()`

**Examples**

```r
null_model(mode = "regression")
```
predict.model_fit  Model predictions

Description

Apply a model to create different types of predictions. predict() can be used for all types of models and used the "type" argument for more specificity.

Usage

```r
## S3 method for class 'model_fit'
predict(object, new_data, type = NULL, opts = list(), ...)
```

Arguments

- `object`: An object of class `model_fit`
- `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.
- `opts`: A list of optional arguments to the underlying predict function that will be used when `type = "raw"`. The list should not include options for the model object or the new data being predicted.
- `...`: Arguments to the underlying model's prediction function cannot be passed here (see `opts`). There are some `parsnip` related options that can be passed, depending on the value of `type`. Possible arguments are:
  - `level`: for types of "conf_int" and "pred_int" this is the parameter for the tail area of the intervals (e.g. confidence level for confidence intervals). Default value is 0.95.
  - `std_error`: add the standard error of fit or prediction for types of "conf_int" and "pred_int". Default value is FALSE.
  - `quantile`: the quantile(s) for quantile regression (not implemented yet)
  - `time`: the time(s) for hazard probability estimates (not implemented yet)

Details

If "type" is not supplied to `predict()`, then a choice is made (type = "numeric" for regression models and type = "class" for classification).

`predict()` is designed to provide a tidy result (see "Value" section below) in a tibble output format.

When using `type = "conf_int"` and `type = "pred_int"`, the options `level` and `std_error` can be used. The latter is a logical for an extra column of standard error values (if available).
Value

With the exception of `type = "raw"`, the results of `predict.model_fit()` will be a tibble as many rows in the output as there are rows in `new_data` and the column names will be predictable.

For numeric results with a single outcome, the tibble will have a `.pred` column and `.pred.Yname` for multivariate results.

For hard class predictions, the column is named `.pred_class` and, when `type = "prob"`, the columns are `.pred_classlevel`.

`type = "conf.int"` and `type = "pred.int"` return tibbles with columns `.pred_lower` and `.pred_upper` with an attribute for the confidence level. In the case where intervals can be produces for class probabilities (or other non-scalar outputs), the columns will be named `.pred_lower_classlevel` and so on.

Quantile predictions return a tibble with a column `.pred`, which is a list-column. Each list element contains a tibble with columns `.pred` and `.quantile` (and perhaps other columns).

Using `type = "raw"` with `predict.model_fit()` will return the unadulterated results of the prediction function.

In the case of Spark-based models, since table columns cannot contain dots, the same convention is used except 1) no dots appear in names and 2) vectors are never returned but type-specific prediction functions.

When the model fit failed and the error was captured, the `predict()` function will return the same structure as above but filled with missing values. This does not currently work for multivariate models.

Examples

```r
library(dplyr)

lm_model <-
  linear_reg() %>%
  set_engine("lm") %>%
  fit(mpg ~ ., data = mtcars %>% slice(11:32))

pred_cars <-
  mtcars %>%
  slice(1:10) %>%
  select(-mpg)

predict(lm_model, pred_cars)

predict(
  lm_model,
  pred_cars,
  type = "conf.int",
  level = 0.90
)

predict(
  lm_model,
  pred_cars,
  type = "raw",
  opts = list(type = "terms")
)
```
rand_forest

Description
rand_forest() is a way to generate a specification of a model before fitting and allows the model to be created using different packages in R or via Spark. The main arguments for the model are:

- mtry: The number of predictors that will be randomly sampled at each split when creating the tree models.
- trees: The number of trees contained in the ensemble.
- min_n: The minimum number of data points in a node that are required for the node to be split further.

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using set_engine(). If left to their defaults here (NULL), the values are taken from the underlying model functions. If parameters need to be modified, update() can be used in lieu of recreating the object from scratch.

Usage
rand_forest(mode = "unknown", mtry = NULL, trees = NULL, min_n = NULL)

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

Arguments
- mode: A single character string for the type of model. Possible values for this model are "unknown", "regression", or "classification".
- 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.
- object: A random forest model specification.
- parameters: A 1-row tibble or named list with main parameters to update. If the individual arguments are used, these will supersede the values in parameters. Also, using engine arguments in this object will result in an error.
- fresh: A logical for whether the arguments should be modified in-place of or replaced wholesale.
- ...: Not used for update().
Details

The model can be created using the `fit()` function using the following `engines`:

- R: "ranger" (the default) or "randomForest"
- Spark: "spark"

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:

**ranger classification**

```r
ranger::ranger(formula = missing_arg(), data = missing_arg(),
                case.weights = missing_arg(), num.threads = 1, verbose = FALSE,
                seed = sample.int(10^5, 1), probability = TRUE)
```

**ranger regression**

```r
ranger::ranger(formula = missing_arg(), data = missing_arg(),
                case.weights = missing_arg(), num.threads = 1, verbose = FALSE,
                seed = sample.int(10^5, 1))
```

**randomForests classification**

```r
randomForest::randomForest(x = missing_arg(), y = missing_arg())
```

**randomForests regression**

```r
randomForest::randomForest(x = missing_arg(), y = missing_arg())
```

**spark classification**

```r
sparklyr::ml_random_forest(x = missing_arg(), formula = missing_arg(),
                           type = "classification", seed = sample.int(10^5, 1))
```

**spark regression**

```r
sparklyr::ml_random_forest(x = missing_arg(), formula = missing_arg(),
                           type = "regression", seed = sample.int(10^5, 1))
```

For **ranger** confidence intervals, the intervals are constructed using the form estimate +/- $z \times \text{std.error}$. For classification probabilities, these values can fall outside of $[0, 1]$ and will be coerced to be in this range.

Note

For models created using the spark engine, there are several differences to consider. First, only the formula interface to `fit()` is available; using `fit.xy()` will generate an error. Second, the predictions will always be in a spark table format. The names will be the same as documented but without the dots. Third, there is no equivalent to factor columns in spark tables so class predictions are returned as character columns. Fourth, to retain the model object for a new R session (via `save`), the `model$fit` element of the `parsnip` object should be serialized via `ml_save(object$fit)` and separately saved to disk. In a new session, the object can be reloaded and reattached to the `parsnip` object.
See Also

[fit()]

Examples

```r
rand_forest(mode = "classification", trees = 2000)
# Parameters can be represented by a placeholder:
rand_forest(mode = "regression", mtry = varying())
model <- rand_forest(mtry = 10, min_n = 3)
model
update(model, mtry = 1)
update(model, mtry = 1, fresh = TRUE)
```

---

### 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

```r
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

```r
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 varying().

Value

An updated model specification.

Examples

# First, set general arguments using the standardized names
mod <- logistic_reg(mixture = 1/3) %>%
  # now say how you want to fit the model and another other options
  set_engine("glmnet", nlambda = 10)
translate(mod, engine = "glmnet")

surv_reg

General Interface for Parametric Survival Models

Description

surv_reg() is a way to generate a specification of a model before fitting and allows the model to be created using R. The main argument for the model is:

- dist: The probability distribution of the outcome.

This argument is converted to its specific names at the time that the model is fit. Other options and argument can be set using set_engine(). If left to its default here (NULL), the value is taken from the underlying model functions.

If parameters need to be modified, this function can be used in lieu of recreating the object from scratch.
Usage

```r
surv_reg(mode = "regression", dist = NULL)
```

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

Arguments

- **mode**
  A single character string for the type of model. The only possible value for this model is "regression".

- **dist**
  A character string for the outcome distribution. "weibull" is the default.

- **object**
  A survival regression model specification.

- **parameters**
  A 1-row tibble or named list with main parameters to update. If the individual arguments are used, these will supersede the values in `parameters`. Also, using engine arguments in this object will result in an error.

- **fresh**
  A logical for whether the arguments should be modified in-place of or replaced wholesale.

- **...**
  Not used for `update()`.

Details

The data given to the function are not saved and are only used to determine the mode of the model. For `surv_reg()`, the mode will always be "regression".

Since survival models typically involve censoring (and require the use of `survival::Surv()` objects), the `fit()` function will require that the survival model be specified via the formula interface.

Also, for the `flexsurv::flexsurvfit` engine, the typical `strata` function cannot be used. To achieve the same effect, the extra parameter roles can be used (as described above).

For `surv_reg()`, the mode will always be "regression".

The model can be created using the `fit()` function using the following engines:

- R: "flexsurv", "survival" (the default)

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:

- **flexsurv**

  ```r
  flexsurv::flexsurvreg(formula = missing_arg(), data = missing_arg(),
                       weights = missing_arg())
  ```

- **survival**

  ```r
  survival::survreg(formula = missing_arg(), data = missing_arg(),
                    weights = missing_arg(), model = TRUE)
  ```

Note that `model = TRUE` is needed to produce quantile predictions when there is a stratification variable and can be overridden in other cases.
References


See Also

`fit()`, `survival::Surv()`

Examples

```r
surv_reg()
# Parameters can be represented by a placeholder:
surv_reg(dist = varying())

model <- surv_reg(dist = "weibull")
model
update(model, dist = "lnorm")
```

---

**svm_poly**

*General interface for polynomial support vector machines*

**Description**

`svm_poly()` is a way to generate a *specification* of a model before fitting and allows the model to be created using different packages in R or via Spark. The main arguments for the model are:

- `cost`: The cost of predicting a sample within or on the wrong side of the margin.
- `degree`: The polynomial degree.
- `scale_factor`: A scaling factor for the kernel.
- `margin`: The epsilon in the SVM insensitive loss function (regression only)

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using `set_engine()`. If left to their defaults here (NULL), the values are taken from the underlying model functions. If parameters need to be modified, `update()` can be used in lieu of recreating the object from scratch.

**Usage**

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

```r
## S3 method for class 'svm_poly'
update(
  object,
  parameters = NULL,
```
Arguments

- **mode**: A single character string for the type of model. Possible values for this model are "unknown", "regression", or "classification".
- **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).
- **object**: A polynomial SVM model specification.
- **parameters**: A 1-row tibble or named list with main parameters to update. If the individual arguments are used, these will supersede the values in `parameters`. Also, using engine arguments in this object will result in an error.
- **fresh**: A logical for whether the arguments should be modified in-place of or replaced wholesale.
- **...**: Not used for `update()`.

Details

The model can be created using the `fit()` function using the following engines:

- **R**: "kernlab" (the default)

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:

**kernlab**

- **classification**

  ```r
  kernlab::ksvm(x = missing_arg(), y = missing_arg(), kernel = "polydot",
               prob.model = TRUE)
  ```

- **regression**

  ```r
  kernlab::ksvm(x = missing_arg(), y = missing_arg(), kernel = "polydot")
  ```

See Also

- `fit()`
Examples

```r
svm_poly(mode = "classification", degree = 1.2)
# Parameters can be represented by a placeholder:
svm_poly(mode = "regression", cost = varying())
model <- svm_poly(cost = 10, scale_factor = 0.1)
model
update(model, cost = 1)
update(model, cost = 1, fresh = TRUE)
```

**svm_rbf**

General interface for radial basis function support vector machines

Description

`svm_rbf()` is a way to generate a specification of a model before fitting and allows the model to be created using different packages in R or via Spark. The main arguments for the model are:

- **cost**: The cost of predicting a sample within or on the wrong side of the margin.
- **rbf_sigma**: The precision parameter for the radial basis function.
- **margin**: The epsilon in the SVM insensitive loss function (regression only)

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using `set_engine()`. If left to their defaults here (NULL), the values are taken from the underlying model functions. If parameters need to be modified, `update()` can be used in lieu of recreating the object from scratch.

Usage

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

### S3 method for class 'svm_rbf'

```r
update(
  object,
  parameters = NULL,
  cost = NULL,
  rbf_sigma = NULL,
  margin = NULL,
  fresh = FALSE,
  ...
)
```

Arguments

- **mode**: A single character string for the type of model. Possible values for this model are "unknown", "regression", or "classification".
- **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)

**object**
A radial basis function SVM model specification.

**parameters**
A 1-row tibble or named list with *main* parameters to update. If the individual arguments are used, these will supersede the values in `parameters`. Also, using engine arguments in this object will result in an error.

**fresh**
A logical for whether the arguments should be modified in-place of or replaced wholesale.

... Not used for `update()`.

**Details**

The model can be created using the `fit()` function using the following *engines*:

- **R**: "kernlab" (the default)

**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:

**kernlab** classification

```r
kernlab::ksvm(x = missing_arg(), y = missing_arg(), kernel = "rbfdot", prob.model = TRUE)
```

**kernlab** regression

```r
ekernlab::ksvm(x = missing_arg(), y = missing_arg(), kernel = "rbfdot")
```

**See Also**

`fit()`

**Examples**

```r
svm_rbf(mode = "classification", rbf_sigma = 0.2)
# Parameters can be represented by a placeholder:
svm_rbf(mode = "regression", cost = varying())
model <- svm_rbf(cost = 10, rbf_sigma = 0.1)
model
update(model, cost = 1)
update(model, cost = 1, fresh = TRUE)
```
**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**  
```r
 tidy.model_fit(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**  
```r
 translate(x, ...) 
```

**Arguments**  
- `x`  
  A model specification.
- `...`  
  Not currently used.

**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 `varying` arguments if these are in the specification. 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

```r
lm_spec <- linear_reg(penalty = 0.01)
# 'penalty' is translated to 'lambda'
translate(lm_spec, engine = "glmnet")
# 'penalty' not applicable for this model.
translate(lm_spec, engine = "lm")
# 'penalty' is translated to 'reg_param'
translate(lm_spec, engine = "spark")
# with a placeholder for an unknown argument value:
translate(linear_reg(mixture = varying()), engine = "glmnet")
```

---

**varying**

* A placeholder function for argument values

**Description**

`varying()` is used when a parameter will be specified at a later date.

**Usage**

`varying()`

---

**varying_args.model_spec**

* Determine varying arguments

**Description**

`varying_args()` takes a model specification or a recipe and returns a tibble of information on all possible varying arguments and whether or not they are actually varying.

**Usage**

```r
## S3 method for class 'model_spec'
varying_args(object, full = TRUE, ...)

## S3 method for class 'recipe'
varying_args(object, full = TRUE, ...)

## S3 method for class 'step'
varying_args(object, full = TRUE, ...)
```
Arguments

- **object**: A `model_spec` or a `recipe`.
- **full**: A single logical. Should all possible varying parameters be returned? If `FALSE`, then only the parameters that are actually varying are returned.
- ...: Not currently used.

Details

The `id` column is determined differently depending on whether a `model_spec` or a `recipe` is used. For a `model_spec`, the first class is used. For a `recipe`, the unique step `id` is used.

Value

A tibble with columns for the parameter name (`name`), whether it contains *any* varying value (`varying`), the `id` for the object (`id`), and the class that was used to call the method (`type`).

Examples

```r
data <- varying_args(model_spec)

# List all possible varying args for the random forest spec
rand_forest() %>% varying_args()

# mtry is now recognized as varying
rand_forest(mtry = varying()) %>% varying_args()

# Even engine specific arguments can vary
rand_forest() %>%
  set_engine("ranger", sample.fraction = varying()) %>%
  varying_args()

# List only the arguments that actually vary
rand_forest() %>%
  set_engine("ranger", sample.fraction = varying()) %>%
  varying_args(full = FALSE)
```

```r
data <- varying_args(model_spec)

# List all possible varying args for the random forest spec
rand_forest() %>% varying_args()

# mtry is now recognized as varying
rand_forest(mtry = varying()) %>% varying_args()

# Even engine specific arguments can vary
rand_forest() %>%
  set_engine("ranger", sample.fraction = varying()) %>%
  varying_args()

# List only the arguments that actually vary
rand_forest() %>%
  set_engine("randomForest",
            strata = Class,
            sampsize = varying())
  varying_args()
```
Index

*Topic models
nullmodel, 29  
cols (descriptors), 9  
dat (descriptors), 9  
lvls (descriptors), 9  
obs (descriptors), 9  
preds (descriptors), 9  
x (descriptors), 9  
y (descriptors), 9  
add_rowindex, 3
boost_tree, 3
control_parsnip, 6
control_parsnip(), 11, 12
decision_tree, 7
descriptors, 9
fit(), 6, 9, 15, 17, 19, 21, 24, 27, 29, 31, 36, 38-40, 42
fit.model_spec, 11
fit_control (control_parsnip), 6
fit_xy(), 24
fit_xy.model_spec (fit.model_spec), 11
linear_reg, 12
logistic_reg, 15
mars, 17
mlp, 19
model_fit, 22
model_spec, 22, 23
multi_predict, 27
multinom_reg, 25
nearest_neighbor, 28
null_model, 31
nullmodel, 29
predict.model_fit, 32
predict.nullmodel (nullmodel), 29
print.nullmodel (nullmodel), 29
rand_forest, 34
set.args, 36
set_engine, 37
set_engine(), 6, 11, 12, 15
set_mode (set_args), 36
surv_reg, 37
survival::Surv(), 38, 39
svm_poly, 39
svm_rbf, 41
tibble::tibble(), 43
tidy.model_fit, 43
translate, 43
update.boost_tree (boost_tree), 3
update.decision_tree (decision_tree), 7
update.linear_reg (linear_reg), 12
update.logistic_reg (logistic_reg), 15
update.mars (mars), 17
update.mlp (mlp), 19
update.multinom_reg (multinom_reg), 25
update.rand_forest (rand_forest), 34
update.surv_reg (surv_reg), 37
update.svm_poly (svm_poly), 39
update.svm_rbf (svm_rbf), 41
varying, 44
varying(), 44
varying.args.model_spec, 44
varying.args.recipe
   (varying.args.model_spec), 44
varying.args.step
   (varying.args.model_spec), 44