Package 'parsnip'

May 27, 2021

Version 0.1.6

Title A Common API to Modeling and Analysis Functions

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', etc).

Maintainer Max Kuhn <max@rstudio.com>

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

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

License MIT + file LICENSE

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 (>= 0.1.0), glue, lifecycle, magrittr, stats, tidyr (>= 1.0.0), globals, prettyunits, vctrs (>= 0.2.0)

RoxygenNote 7.1.1.9001

Suggests testthat, knitr, rmarkdown, survival, keras, xgboost, covr, C50, sparklyr (>= 1.0.0), earth, kernlab, kknn, randomForest, ranger (>= 0.12.0), rpart, MASS, nlme, modeldata, LiblineaR, Matrix

NeedsCompilation no

Author Max Kuhn [aut, cre], Davis Vaughan [aut], Emil Hvitfeldt [ctb], RStudio [cph]

Repository CRAN

Date/Publication 2021-05-27 05:20:02 UTC

R topics documented:

`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, ...)
```

Arguments

x	A `model_fit` object produced by `fit()` or `fit_xy()`.
new_data	A data frame or matrix.
...	Not currently used.

Details

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

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.

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])
```

`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 is 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.
- `stop_iter`: The number of iterations without improvement before stopping.

These arguments are converted to their specific names at the time that the model is fit. Other options and arguments 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,
  stop_iter = NULL
)
```

Arguments

`mode`	A single character string for the type of model. 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 (`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 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) (`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	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 (xgboost only).

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"

For this model, other packages may add additional engines. Use show_engines() to see the current set of engines.

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:

xgboost:

```
boost_tree() %>%
  set_engine("xgboost") %>%
  set_mode("regression") %>%
  translate()
```

```
## Boosted Tree Model Specification (regression)
##
## Computational engine: xgboost
##
## Model fit template:
## parsnip::xgb_train(x = missing_arg(), y = missing_arg(), nthread = 1,
##     verbose = 0)
```

```
boost_tree() %>%
  set_engine("xgboost") %>%
  set_mode("classification") %>%
  translate()
```

```
## Boosted Tree Model Specification (classification)
##
## Computational engine: xgboost
##
## Model fit template:
## parsnip::xgb_train(x = missing_arg(), y = missing_arg(), nthread = 1,
##     verbose = 0)
```

Note that, for most engines to `boost_tree()`, the `sample_size` argument is in terms of the *number* of training set points. The `xgboost` package parameterizes this as the *proportion* of training set samples instead. When using the `tune`, this **occurs automatically**.

If you would like to use a custom range when tuning `sample_size`, the `dials::sample_prop()` function can be used in that case. For example, using a parameter set:

```
mod <-
  boost_tree(sample_size = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

# update the parameters using the `dials` function
mod_param <-
  mod %>%
  parameters() %>%
  update(sample_size = sample_prop(c(0.4, 0.9)))
```

For this engine, tuning over `trees` is very efficient since the same model object can be used to make predictions over multiple values of `trees`.

Note that `xgboost` models require that non-numeric predictors (e.g., factors) must be converted to dummy variables or some other numeric representation. By default, when using `fit()` with `xgboost`, a one-hot encoding is used to convert factor predictors to indicator variables.

Finally, in the classification mode, non-numeric outcomes (i.e., factors) are converted to numeric. For binary classification, the `event_level` argument of `set_engine()` can be set to either `"first"` or `"second"` to specify which level should be used as the event. This can be helpful when a watchlist is used to monitor performance from with the xgboost training process.

C5.0:

```
boost_tree() %>%
  set_engine("C5.0") %>%
  set_mode("classification") %>%
  translate()
```

```
## Boosted Tree Model Specification (classification)
##
## Computational engine: C5.0
##
## Model fit template:
## parsnip::C5.0_train(x = missing_arg(), y = missing_arg(), weights = missing_arg())
```

Note that `C50::C5.0()` does not require factor predictors to be converted to indicator variables. `fit()` does not affect the encoding of the predictor values (i.e. factors stay factors) for this model.

For this engine, tuning over `trees` is very efficient since the same model object can be used to make predictions over multiple values of `trees`.

spark:

```
boost_tree() %>%
  set_engine("spark") %>%
  set_mode("regression") %>%
  translate()
```

```
## Boosted Tree Model Specification (regression)
##
## Computational engine: spark
##
## Model fit template:
## sparklyr::ml_gradient_boosted_trees(x = missing_arg(), formula = missing_arg(),
##     type = "regression", seed = sample.int(10^5, 1))
```

```
boost_tree() %>%
  set_engine("spark") %>%
  set_mode("classification") %>%
  translate()
```

```
## Boosted Tree Model Specification (classification)
##
## Computational engine: spark
##
## Model fit template:
## sparklyr::ml_gradient_boosted_trees(x = missing_arg(), formula = missing_arg(),
##     type = "classification", seed = sample.int(10^5, 1))
```

`fit()` does not affect the encoding of the predictor values (i.e. factors stay factors) for this model.

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	xgboost	C5.0	spark
tree_depth	max_depth (6)	NA	max_depth (5)
trees	nrounds (15)	trials (15)	max_iter (20)
learn_rate	eta (0.3)	NA	step_size (0.1)
mtry	colsample_bynode (character(0))	NA	feature_subset_strategy (see below)
min_n	min_child_weight (1)	minCases (2)	min_instances_per_node (1)
loss_reduction	gamma (0)	NA	min_info_gain (0)
sample_size	subsample (1)	sample (0)	subsampling_rate (1)
stop_iter	early_stop (NULL)	NA	NA

For spark, the default `mtry` is the square root of the number of predictors for classification, and one-third of the predictors for regression.

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(), update()

Examples

```
show_engines("boost_tree")

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

---

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

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.

`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 arguments 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
)
```

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).
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

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 below:

rpart:

```
decision_tree() %>%
  set_engine("rpart") %>%
  set_mode("regression") %>%
  translate()

## Decision Tree Model Specification (regression)
##
## Computational engine: rpart
##
## Model fit template:
## rpart::rpart(formula = missing_arg(), data = missing_arg(), weights = missing_arg())
```

```
decision_tree() %>%
  set_engine("rpart") %>%
  set_mode("classification") %>%
  translate()
```

```
## Decision Tree Model Specification (classification)
##
## Computational engine: rpart
##
## Model fit template:
## rpart::rpart(formula = missing_arg(), data = missing_arg(), weights = missing_arg())
```

Note that `rpart::rpart()` does not require factor predictors to be converted to indicator variables. `fit()` does not affect the encoding of the predictor values (i.e. factors stay factors) for this model

C5.0:

```
decision_tree() %>%
  set_engine("C5.0") %>%
  set_mode("classification") %>%
  translate()
```

```
## Decision Tree Model Specification (classification)
##
## Computational engine: C5.0
##
## Model fit template:
## parsnip::C5.0_train(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
##     trials = 1)
```

Note that `C50::C5.0()` does not require factor predictors to be converted to indicator variables. `fit()` does not affect the encoding of the predictor values (i.e. factors stay factors) for this model

spark:

```
decision_tree() %>%
  set_engine("spark") %>%
  set_mode("regression") %>%
  translate()
```

```
## Decision Tree Model Specification (regression)
##
## Computational engine: spark
##
## Model fit template:
## sparklyr::ml_decision_tree_regressor(x = missing_arg(), formula = missing_arg(),
##     seed = sample.int(10^5, 1))
```

```
decision_tree() %>%
  set_engine("spark") %>%
  set_mode("classification") %>%
  translate()
```

```
## Decision Tree Model Specification (classification)
##
## Computational engine: spark
##
## Model fit template:
## sparklyr::ml_decision_tree_classifier(x = missing_arg(), formula = missing_arg(),
##     seed = sample.int(10^5, 1))
```

`fit()` does not affect the encoding of the predictor values (i.e. factors stay factors) for this model

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	rpart	C5.0	spark
tree_depth	maxdepth (30)	NA	max_depth (5)
min_n	minsplit (20)	minCases (2)	min_instances_per_node (1)
cost_complexity	cp (0.01)	NA	NA

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()`, `update()`

Examples

```
show_engines("decision_tree")

decision_tree(mode = "classification", tree_depth = 5)
# Parameters can be represented by a placeholder:
decision_tree(mode = "regression", cost_complexity = varying())
```

`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.

`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, 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, 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.

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
```

---

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

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 mixture amounts of different types of regularization (see below). 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 arguments 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

```
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`	A 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`; see below).
`mixture`	A number between zero and one (inclusive) that is the proportion of L1 regularization (i.e. lasso) in the model. When `mixture = 1`, it is a pure lasso model while `mixture = 0` indicates that ridge regression is being used. (`glmnet` and `spark` only).

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"`

For this model, other packages may add additional engines. Use `show_engines()` to see the current set of engines.

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.

lm:

```
linear_reg() %>%
  set_engine("lm") %>%
  translate()
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
##
## Model fit template:
## stats::lm(formula = missing_arg(), data = missing_arg(), weights = missing_arg())
```

glmnet:

```
linear_reg(penalty = 0.1) %>%
  set_engine("glmnet") %>%
  translate()
```

```
## Linear Regression Model Specification (regression)
##
## Main Arguments:
##   penalty = 0.1
##
## Computational engine: glmnet
##
## Model fit template:
## glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
##     family = "gaussian")
```

The glmnet engine requires a single value for the `penalty` argument (a number or `tune()`), but the full regularization path is always fit regardless of the value given to `penalty`. To pass in a custom sequence of values for glmnet's `lambda`, use the argument `path_values` in `set_engine()`. This will assign the value of the glmnet `lambda` parameter without disturbing the value given of `linear_reg(penalty)`. For example:

```
linear_reg(penalty = .1) %>%
  set_engine("glmnet", path_values = c(0, 10^seq(-10, 1, length.out = 20))) %>%
  translate()
```

```
## Linear Regression Model Specification (regression)
##
## Main Arguments:
##   penalty = 0.1
##
## Computational engine: glmnet
##
## Model fit template:
```

```
## glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
##     lambda = c(0, 10^seq(-10, 1, length.out = 20)), family = "gaussian")
```

When fitting a pure ridge regression model (i.e., `penalty = 0`), we *strongly suggest* that you pass in a vector for `path_values` that includes zero. See issue #431 for a discussion.

When using `predict()`, the single `penalty` value used for prediction is the one specified in `linear_reg()`.

To predict on multiple penalties, use the `multi_predict()` function. This function returns a tibble with a list column called `.pred` containing all of the penalty results.

stan:

```
linear_reg() %>%
  set_engine("stan") %>%
  translate()
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: stan
##
## Model fit template:
## rstanarm::stan_glm(formula = missing_arg(), data = missing_arg(),
##     weights = missing_arg(), family = stats::gaussian, refresh = 0)
```

Note that the `refresh` default prevents logging of the estimation process. Change this value in `set_engine()` to show the logs.

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.

spark:

```
linear_reg() %>%
  set_engine("spark") %>%
  translate()
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: spark
##
## Model fit template:
## sparklyr::ml_linear_regression(x = missing_arg(), formula = missing_arg(),
##     weight_col = missing_arg())
```

keras:

```
linear_reg() %>%
  set_engine("keras") %>%
  translate()
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: keras
##
## Model fit template:
## parsnip::keras_mlp(x = missing_arg(), y = missing_arg(), hidden_units = 1,
##     act = "linear")
```

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	glmnet	spark	keras
penalty	lambda	reg_param (0)	penalty (0)
mixture	alpha (1)	elastic_net_param (0)	NA

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()`, `update()`

Examples

```
show_engines("linear_reg")

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

`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 mixture amounts of different types of regularization (see below). 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 arguments 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

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

Arguments

`mode`	A single character string for the type of model. The only possible value for this model is "classification".
`penalty`	A non-negative number representing the total amount of regularization (`glmnet`, `LiblineaR`, `keras`, and `spark` 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) that is the proportion of L1 regularization (i.e. lasso) in the model. When `mixture = 1`, it is a pure lasso model while `mixture = 0` indicates that ridge regression is being used. (`glmnet`, `LiblineaR`, and `spark` only). For `LiblineaR` models, `mixture` must be exactly 0 or 1 only.

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), `"glmnet"`, or `"LiblineaR"`
- **Stan**: `"stan"`
- **Spark**: `"spark"`
- **keras**: `"keras"`

For this model, other packages may add additional engines. Use `show_engines()` to see the current set of engines.

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.

glm:

```
logistic_reg() %>%
  set_engine("glm") %>%
  translate()
```

```
## Logistic Regression Model Specification (classification)
##
## Computational engine: glm
##
## Model fit template:
## stats::glm(formula = missing_arg(), data = missing_arg(), weights = missing_arg(),
##     family = stats::binomial)
```

glmnet:

```
logistic_reg(penalty = 0.1) %>%
  set_engine("glmnet") %>%
  translate()
```

```
## Logistic Regression Model Specification (classification)
##
## Main Arguments:
##   penalty = 0.1
##
## Computational engine: glmnet
##
## Model fit template:
## glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
##     family = "binomial")
```

The glmnet engine requires a single value for the `penalty` argument (a number or `tune()`), but the full regularization path is always fit regardless of the value given to `penalty`. To pass in a custom sequence of values for glmnet's `lambda`, use the argument `path_values` in `set_engine()`. This will assign the value of the glmnet `lambda` parameter without disturbing the value given of `logistic_reg(penalty)`. For example:

```
logistic_reg(penalty = .1) %>%
  set_engine("glmnet", path_values = c(0, 10^seq(-10, 1, length.out = 20))) %>%
  translate()
```

```
## Logistic Regression Model Specification (classification)
##
## Main Arguments:
##   penalty = 0.1
##
## Computational engine: glmnet
##
```

```
## Model fit template:
## glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
##     lambda = c(0, 10^seq(-10, 1, length.out = 20)), family = "binomial")
```

When fitting a pure ridge regression model (i.e., `penalty = 0`), we *strongly suggest* that you pass in a vector for `path_values` that includes zero. See issue #431 for a discussion.

When using `predict()`, the single `penalty` value used for prediction is the one specified in `logistic_reg()`.

To predict on multiple penalties, use the `multi_predict()` function. This function returns a tibble with a list column called `.pred` containing all of the penalty results.

LiblineaR:

```
logistic_reg() %>%
  set_engine("LiblineaR") %>%
  translate()
```

```
## Logistic Regression Model Specification (classification)
##
## Computational engine: LiblineaR
##
## Model fit template:
## LiblineaR::LiblineaR(x = missing_arg(), y = missing_arg(), wi = missing_arg(),
##     verbose = FALSE)
```

For `LiblineaR` models, the value for `mixture` can either be 0 (for ridge) or 1 (for lasso) but not other intermediate values. In the `LiblineaR` documentation, these correspond to types 0 (L2-regularized) and 6 (L1-regularized).

Be aware that the `LiblineaR` engine regularizes the intercept. Other regularized regression models do not, which will result in different parameter estimates.

stan:

```
logistic_reg() %>%
  set_engine("stan") %>%
  translate()
```

```
## Logistic Regression Model Specification (classification)
##
## Computational engine: stan
##
## Model fit template:
## rstanarm::stan_glm(formula = missing_arg(), data = missing_arg(),
##     weights = missing_arg(), family = stats::binomial, refresh = 0)
```

Note that the `refresh` default prevents logging of the estimation process. Change this value in `set_engine()` to show the logs.

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.

spark:

```
logistic_reg() %>%
  set_engine("spark") %>%
  translate()
```

```
## Logistic Regression Model Specification (classification)
##
## Computational engine: spark
##
## Model fit template:
## sparklyr::ml_logistic_regression(x = missing_arg(), formula = missing_arg(),
##     weight_col = missing_arg(), family = "binomial")
```

keras:

```
logistic_reg() %>%
  set_engine("keras") %>%
  translate()
```

```
## Logistic Regression Model Specification (classification)
##
## Computational engine: keras
##
## Model fit template:
## parsnip::keras_mlp(x = missing_arg(), y = missing_arg(), hidden_units = 1,
##     act = "linear")
```

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	glmnet	LiblineaR	spark	keras
penalty	lambda	cost	reg_param (0)	penalty (0)
mixture	alpha (1)	type (0)	elastic_net_param (0)	NA

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(), update()

Examples

```
show_engines("logistic_reg")

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

---

mars *General Interface for MARS*

Description

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 an 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 arguments 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
)
```

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.

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 below.

earth:

```
mars() %>%
  set_engine("earth") %>%
  set_mode("regression") %>%
  translate()
```

```
## MARS Model Specification (regression)
##
## Computational engine: earth
##
## Model fit template:
## earth::earth(formula = missing_arg(), data = missing_arg(), weights = missing_arg(),
##     keepxy = TRUE)
```

```
mars() %>%
  set_engine("earth") %>%
  set_mode("classification") %>%
  translate()
```

```
## MARS Model Specification (classification)
##
## Engine-Specific Arguments:
##   glm = list(family = stats::binomial)
##
## Computational engine: earth
##
## Model fit template:
## earth::earth(formula = missing_arg(), data = missing_arg(), weights = missing_arg(),
##     glm = list(family = stats::binomial), 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.

Also, `fit()` passes the data directly to `earth::earth()` so that its formula method can create dummy variables as-needed.

For this engine, tuning over `num_terms` is very efficient since the same model object can be used to make predictions over multiple values of `num_terms`.

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	earth
num_terms	nprune
prod_degree	degree (1)
prune_method	pmethod (backward)

See Also

`fit()`, `set_engine()`, `update()`

Examples

```
show_engines("mars")

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

`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 *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).

Usage

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

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"

Details

These arguments are converted to their specific names at the time that the model is fit. Other options and arguments 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"`

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:

keras:

```
mlp() %>%
  set_engine("keras") %>%
  set_mode("regression") %>%
  translate()
```

```
## Single Layer Neural Network Specification (regression)
##
## Computational engine: keras
##
## Model fit template:
## parsnip::keras_mlp(x = missing_arg(), y = missing_arg())
```

```
mlp() %>%
  set_engine("keras") %>%
  set_mode("classification") %>%
  translate()
```

```
## Single Layer Neural Network Specification (classification)
##
## Computational engine: keras
##
## Model fit template:
## parsnip::keras_mlp(x = missing_arg(), y = missing_arg())
```

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

nnet:

```
mlp() %>%
  set_engine("nnet") %>%
  set_mode("regression") %>%
  translate()
```

```
## Single Layer Neural Network Specification (regression)
##
## Main Arguments:
##   hidden_units = 5
##
## Computational engine: nnet
##
## Model fit template:
## nnet::nnet(formula = missing_arg(), data = missing_arg(), weights = missing_arg(),
##     size = 5, trace = FALSE, linout = TRUE)
```

```
mlp() %>%
  set_engine("nnet") %>%
  set_mode("classification") %>%
  translate()
```

```
## Single Layer Neural Network Specification (classification)
##
## Main Arguments:
##   hidden_units = 5
##
## Computational engine: nnet
##
## Model fit template:
## nnet::nnet(formula = missing_arg(), data = missing_arg(), weights = missing_arg(),
##     size = 5, trace = FALSE, linout = FALSE)
```

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	keras	nnet
hidden_units	hidden_units (5)	size
penalty	penalty (0)	decay (0)
dropout	dropout (0)	NA
epochs	epochs (20)	maxit (100)
activation	activation (softmax)	NA

See Also

fit(), set_engine(), update()

Examples

```
show_engines("mlp")
```

```
mlp(mode = "classification", penalty = 0.01)
# Parameters can be represented by a placeholder:
```

```
mlp(mode = "regression", hidden_units = varying())
```

`model_fit` *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 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_spec` *Model Specification Information*

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 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 `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

```
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()` 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

```
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://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 mixture amounts of different types of regularization (see below). 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 arguments 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

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

Arguments

mode	A single character string for the type of model. The only possible value for this model is "classification".
penalty	A 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 is the proportion of L1 regularization (i.e. lasso) in the model. When mixture = 1, it is a pure lasso model while mixture = 0 indicates that ridge regression is being used. (glmnet and spark only).

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"
- **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 below.

glmnet:

```
multinom_reg(penalty = 0.1) %>%
  set_engine("glmnet") %>%
  translate()
```

```
## Multinomial Regression Model Specification (classification)
##
## Main Arguments:
##   penalty = 0.1
##
## Computational engine: glmnet
##
## Model fit template:
## glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
##     family = "multinomial")
```

The glmnet engine requires a single value for the `penalty` argument (a number or `tune()`), but the full regularization path is always fit regardless of the value given to `penalty`. To pass in a custom sequence of values for glmnet's `lambda`, use the argument `path_values` in `set_engine()`. This will assign the value of the glmnet `lambda` parameter without disturbing the value given of `multinom_reg(penalty)`. For example:

```
multinom_reg(penalty = .1) %>%
  set_engine("glmnet", path_values = c(0, 10^seq(-10, 1, length.out = 20))) %>%
  translate()
```

```
## Multinomial Regression Model Specification (classification)
##
## Main Arguments:
##   penalty = 0.1
##
## Computational engine: glmnet
##
## Model fit template:
## glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
##     lambda = c(0, 10^seq(-10, 1, length.out = 20)), family = "multinomial")
```

When fitting a pure ridge regression model (i.e., `penalty = 0`), we *strongly suggest* that you pass in a vector for `path_values` that includes zero. See issue #431 for a discussion.

When using `predict()`, the single `penalty` value used for prediction is the one specified in `multinom_reg()`.

To predict on multiple penalties, use the `multi_predict()` function. This function returns a tibble with a list column called `.pred` containing all of the penalty results.

nnet:

```
multinom_reg() %>%
  set_engine("nnet") %>%
  translate()
```

```
## Multinomial Regression Model Specification (classification)
##
## Computational engine: nnet
##
## Model fit template:
## nnet::multinom(formula = missing_arg(), data = missing_arg(),
##     weights = missing_arg(), trace = FALSE)
```

spark:

```
multinom_reg() %>%
  set_engine("spark") %>%
  translate()
```

```
## Multinomial Regression Model Specification (classification)
##
## Computational engine: spark
##
## Model fit template:
## sparklyr::ml_logistic_regression(x = missing_arg(), formula = missing_arg(),
##     weight_col = missing_arg(), family = "multinomial")
```

keras:

```
multinom_reg() %>%
  set_engine("keras") %>%
  translate()
```

```
## Multinomial Regression Model Specification (classification)
##
## Computational engine: keras
##
## Model fit template:
## parsnip::keras_mlp(x = missing_arg(), y = missing_arg(), hidden_units = 1,
##     act = "linear")
```

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	glmnet	spark	keras	nnet
penalty	lambda	reg_param (0)	penalty (0)	decay (0)
mixture	alpha (1)	elastic_net_param (0)	NA	NA

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()`, `update()`

Examples

```
show_engines("multinom_reg")

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

---

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	A 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).

---

nearest_neighbor *General Interface for K-Nearest Neighbor Models*

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 arguments 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

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

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 below:

kknn:

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

```
## K-Nearest Neighbor Model Specification (regression)
##
## Computational engine: kknn
##
## Model fit template:
## kknn::train.kknn(formula = missing_arg(), data = missing_arg(),
##     ks = min_rows(5, data, 5))
```

```
nearest_neighbor() %>%
  set_engine("kknn") %>%
  set_mode("classification") %>%
  translate()
```

```
## K-Nearest Neighbor Model Specification (classification)
##
## Computational engine: kknn
##
## Model fit template:
## kknn::train.kknn(formula = missing_arg(), data = missing_arg(),
##     ks = min_rows(5, data, 5))
```

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

For this engine, tuning over `neighbors` is very efficient since the same model object can be used to make predictions over multiple values of `neighbors`.

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	kknn
neighbors	ks
weight_func	kernel (optimal)
dist_power	distance (2)

See Also

`fit()`, `set_engine()`, `update()`

Examples

```
show_engines("nearest_neighbor")

nearest_neighbor(neighbors = 11)
```

`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

```
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 below:

parsnip:

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

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

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

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

See Also

`fit()`

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()
```

rand_forest *General Interface for Random Forest Models*

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)
```

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.

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 below:

ranger:

```
rand_forest() %>%
  set_engine("ranger") %>%
  set_mode("regression") %>%
  translate()
```

```
## Random Forest Model Specification (regression)
##
## Computational engine: ranger
##
## Model fit template:
## ranger::ranger(x = missing_arg(), y = missing_arg(), case.weights = missing_arg(),
##     num.threads = 1, verbose = FALSE, seed = sample.int(10^5,
##         1))
```

```
rand_forest() %>%
  set_engine("ranger") %>%
  set_mode("classification") %>%
  translate()
```

```
## Random Forest Model Specification (classification)
##
## Computational engine: ranger
##
## Model fit template:
## ranger::ranger(x = missing_arg(), y = missing_arg(), case.weights = missing_arg(),
##     num.threads = 1, verbose = FALSE, seed = sample.int(10^5,
##         1), probability = TRUE)
```

Note that `ranger::ranger()` does not require factor predictors to be converted to indicator variables. `fit()` does not affect the encoding of the predictor values (i.e. factors stay factors) for this model.

For `ranger` confidence intervals, the intervals are constructed using the form estimate +/- z * std_error. For classification probabilities, these values can fall outside of [0, 1] and will be coerced to be in this range.

randomForest:

```
rand_forest() %>%
  set_engine("randomForest") %>%
  set_mode("regression") %>%
  translate()
```

```
## Random Forest Model Specification (regression)
##
## Computational engine: randomForest
##
## Model fit template:
## randomForest::randomForest(x = missing_arg(), y = missing_arg())
```

```
rand_forest() %>%
  set_engine("randomForest") %>%
  set_mode("classification") %>%
  translate()
```

```
## Random Forest Model Specification (classification)
##
## Computational engine: randomForest
##
## Model fit template:
## randomForest::randomForest(x = missing_arg(), y = missing_arg())
```

Note that `randomForest::randomForest()` does not require factor predictors to be converted to indicator variables. `fit()` does not affect the encoding of the predictor values (i.e. factors stay factors) for this model.

spark:

```
rand_forest() %>%
  set_engine("spark") %>%
  set_mode("regression") %>%
  translate()
```

```
## Random Forest Model Specification (regression)
##
## Computational engine: spark
##
## Model fit template:
## sparklyr::ml_random_forest(x = missing_arg(), formula = missing_arg(),
##     type = "regression", seed = sample.int(10^5, 1))
```

```
rand_forest() %>%
  set_engine("spark") %>%
  set_mode("classification") %>%
  translate()
```

```
## Random Forest Model Specification (classification)
##
## Computational engine: spark
##
```

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

`fit()` does not affect the encoding of the predictor values (i.e. factors stay factors) for this model.

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	ranger	randomForest	spark
mtry	mtry (see below)	mtry (see below)	feature_subset_strategy (see below)
trees	num.trees (500)	ntree (500)	num_trees (20)
min_n	min.node.size (see below)	nodesize (see below)	min_instances_per_node (1)

- For randomForest and spark, the default `mtry` is the square root of the number of predictors for classification, and one-third of the predictors for regression.
- For ranger, the default `mtry` is the square root of the number of predictors.
- The default `min_n` for both ranger and randomForest is 1 for classification and 5 for regression.

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()`, `update()`

Examples

```
show_engines("rand_forest")

rand_forest(mode = "classification", trees = 2000)
# Parameters can be represented by a placeholder:
rand_forest(mode = "regression", mtry = varying())
```

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 relavant 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
```

req_pkgs *Determine required packages for a model*

Description

Determine required packages for a model

Usage

```
req_pkgs(x, ...)

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

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

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

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

Arguments

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

Details

For a model specification, the engine must be set. The list produced by `req_pkgs()`does not include the `parsnip` package while `required_pkgs()` does.

Value

A character string of package names (if any).

Examples

```
should_fail <- try(req_pkgs(linear_reg()), silent = TRUE)
should_fail

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

linear_reg() %>%
  set_engine("lm") %>%
```

```
  fit(mpg ~ ., data = mtcars) %>%
  req_pkgs()
```

`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 `varying()`.

Value

An updated model specification.

Examples

```
# First, set general arguments using the standardized names
mod <-
  logistic_reg(penalty = 0.01, 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")
```

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`-adjacent packages add engines to existing models. For example, the `multilevelmod` package adds additional engines for the `linear_reg()` model and these are not available unless `multilevelmod` 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")
```

---

svm_linear *General interface for linear support vector machines*

Description

svm_linear() 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.
- 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 arguments 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

```
svm_linear(mode = "unknown", cost = NULL, margin = 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
margin	A positive number for the epsilon in the SVM insensitive loss function (regression only)

Details

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

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

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:

LiblineaR:

```
svm_linear() %>%
  set_engine("LiblineaR") %>%
  set_mode("regression") %>%
  translate()
```

```
## Linear Support Vector Machine Specification (regression)
##
## Computational engine: LiblineaR
##
## Model fit template:
## LiblineaR::LiblineaR(x = missing_arg(), y = missing_arg(), wi = missing_arg(),
##     type = 11, svr_eps = 0.1)
```

```
svm_linear() %>%
  set_engine("LiblineaR") %>%
  set_mode("classification") %>%
  translate()
```

```
## Linear Support Vector Machine Specification (classification)
##
## Computational engine: LiblineaR
##
## Model fit template:
## LiblineaR::LiblineaR(x = missing_arg(), y = missing_arg(), wi = missing_arg(),
##     type = 1)
```

Note that the `LiblineaR` engine cannot produce class probabilities. When optimizing the model using the `tune` package, the default metrics require class probabilities. To be able to use the tune_*() functions, a metric set must be passed as an argument and it can only contain metrics associated with hard class predictions (e.g., accuracy and so on).

This engine fits models that are L2-regularized for L2-loss. In the `LiblineaR` documentation, these are types 1 (classification) and 11 (regression).

kernlab:

```
svm_linear() %>%
  set_engine("kernlab") %>%
  set_mode("regression") %>%
  translate()
```

```
## Linear Support Vector Machine Specification (regression)
##
## Computational engine: kernlab
##
## Model fit template:
## kernlab::ksvm(x = missing_arg(), data = missing_arg(), kernel = "vanilladot")
```

```
svm_linear() %>%
  set_engine("kernlab") %>%
  set_mode("classification") %>%
  translate()
```

```
## Linear Support Vector Machine Specification (classification)
##
## Computational engine: kernlab
##
## Model fit template:
## kernlab::ksvm(x = missing_arg(), data = missing_arg(), kernel = "vanilladot",
##     prob.model = TRUE)
```

`fit()` passes the data directly to `kernlab::ksvm()` so that its formula method can create dummy variables as-needed.

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	LiblineaR	kernlab
cost	C (1)	C (1)
margin	svr_eps (0.1)	epsilon (0.1)

See Also

`fit()`, `set_engine()`, `update()`

Examples

```
show_engines("svm_linear")

svm_linear(mode = "classification")
# Parameters can be represented by a placeholder:
svm_linear(mode = "regression", cost = varying())
```

`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 arguments 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

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

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 below:

kernlab:

```
svm_poly() %>%
  set_engine("kernlab") %>%
  set_mode("regression") %>%
  translate()
```

```
## Polynomial Support Vector Machine Specification (regression)
##
## Computational engine: kernlab
##
## Model fit template:
## kernlab::ksvm(x = missing_arg(), data = missing_arg(), kernel = "polydot")
```

```
svm_poly() %>%
  set_engine("kernlab") %>%
  set_mode("classification") %>%
  translate()
```

```
## Polynomial Support Vector Machine Specification (classification)
##
## Computational engine: kernlab
##
## Model fit template:
## kernlab::ksvm(x = missing_arg(), data = missing_arg(), kernel = "polydot",
##     prob.model = TRUE)
```

`fit()` passes the data directly to `kernlab::ksvm()` so that its formula method can create dummy variables as-needed.

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	kernlab
cost	C (1)
degree	degree (1)
scale_factor	scale (1)
margin	epsilon (0.1)

See Also

`fit()`, `set_engine()`, `update()`

Examples

```
show_engines("svm_poly")
```

```
svm_poly(mode = "classification", degree = 1.2)
# Parameters can be represented by a placeholder:
svm_poly(mode = "regression", cost = varying())
```

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 arguments 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

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

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 below:

kernlab:

```
svm_rbf() %>%
  set_engine("kernlab") %>%
  set_mode("regression") %>%
  translate()
```

```
## Radial Basis Function Support Vector Machine Specification (regression)
##
## Computational engine: kernlab
##
## Model fit template:
## kernlab::ksvm(x = missing_arg(), data = missing_arg(), kernel = "rbfdot")
```

```
svm_rbf() %>%
  set_engine("kernlab") %>%
  set_mode("classification") %>%
  translate()
```

```
## Radial Basis Function Support Vector Machine Specification (classification)
##
## Computational engine: kernlab
##
## Model fit template:
## kernlab::ksvm(x = missing_arg(), data = missing_arg(), kernel = "rbfdot",
##     prob.model = TRUE)
```

`fit()` passes the data directly to `kernlab::ksvm()` so that its formula method can create dummy variables as-needed.

Parameter translations:

The standardized parameter names in parsnip can be mapped to their original names in each engine that has main parameters. Each engine typically has a different default value (shown in parentheses) for each parameter.

parsnip	kernlab	liquidSVM
cost	C (1)	lambdas
rbf_sigma	sigma (varies)	gammas
margin	epsilon (0.1)	NA

See Also

`fit()`, `set_engine()`, `update()`

Examples

```
show_engines("svm_rbf")
```

```
svm_rbf(mode = "classification", rbf_sigma = 0.2)
# Parameters can be represented by a placeholder:
svm_rbf(mode = "regression", cost = varying())
```

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 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

```
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 = varying(), mixture = varying()), engine = "glmnet")
```

---

update.boost_tree *Update 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 '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 'decision_tree'
update(
  object,
  parameters = NULL,
  cost_complexity = NULL,
  tree_depth = NULL,
  min_n = 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,
  fresh = FALSE,
  ...
)

## S3 method for class 'multinom_reg'
update(
  object,
  parameters = NULL,
  penalty = NULL,
  mixture = 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 '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 '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.
mtry	A 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 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) (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	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 (xgboost only).
fresh	A logical for whether the arguments should be modified in-place or replaced wholesale.
...	Not used for update().
cost_complexity	A positive number for the the cost/complexity parameter (a.k.a. Cp) used by CART models (rpart only).
penalty	A 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; see below).
mixture	A number between zero and one (inclusive) that is the proportion of L1 regularization (i.e. lasso) in the model. When mixture = 1, it is a pure lasso model while mixture = 0 indicates that ridge regression is being used. (glmnet and spark only).
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.
hidden_units	An integer for the number of units in the hidden model.
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"
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.
dist	A character string for the outcome distribution. "weibull" is the default.
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 <- 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)
```

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

```
## 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

```
# 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)

rand_forest() %>%
  set_engine(
    "randomForest",
    strata = Class,
    sampsize = varying()
  ) %>%
  varying_args()
```

Index