Package 'rules'

January 16, 2021

Title Model Wrappers for Rule-Based Models

Version 0.1.1

Description Bindings for additional models for use with the 'parsnip' package. Models include prediction rule ensembles (Friedman and Popescu, 2008) <doi:10.1214/07-AOAS148>, C5.0 rules (Quinlan, 1992 ISBN: 1558602380), and Cubist (Kuhn and Johnson, 2013) <doi:10.1007/978-1-4614-6849-3>.

License MIT + file LICENSE

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

Depends parsnip (>= 0.1.4)

Suggests testthat, C50, Cubist, xrf (>= 0.2.0), covr, modeldata, spelling, recipes, knitr, rmarkdown

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1.9000

Imports purrr, rlang, tibble, dials, tidyr, dplyr, stringr, generics (>= 0.1.0)

Language en-US

VignetteBuilder knitr

NeedsCompilation no

Author Max Kuhn [aut, cre] (<https://orcid.org/0000-0003-2402-136X>), RStudio [cph]

Maintainer Max Kuhn <max@rstudio.com>

Repository CRAN

Date/Publication 2021-01-16 15:50:02 UTC

R topics documented:

`C5_rules` *General Interface for C5.0 Rule-Based Classification Models*

Description

`C5_rules()` is a way to generate a *specification* of a model before fitting. The main arguments for the model are:

- `trees`: The number of sequential models included in the ensemble (rules are derived from an initial set of boosted trees).
- `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 `parsnip::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

```
C5_rules(mode = "classification", trees = NULL, min_n = NULL)

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

Arguments

`mode`	A single character string for the type of model. The only possible value for this model is "classification".
`trees`	A non-negative integer (no greater than 100 for the number of members of the ensemble.
`min_n`	An integer greater than one zero and nine for the minimum number of data points in a node that are required for the node to be split further.
`object`	A `C5_rules` model specification.

parameters	A 1-row tibble or named list with *main* parameters to update. If the individual arguments are used, these will supersede the values in parameters. Also, using engine arguments in this object will result in an error.
fresh	A logical for whether the arguments should be modified in-place or replaced wholesale.
...	Not used for update().

Details

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

The two main tuning parameters are the number of trees in the boosting ensemble (trees) and the number of samples required to continue splitting when creating a tree (min_n). There are no arguments to control the total number of rules in the ensemble.

Note that C5_rules() does not require that categorical predictors be converted to numeric indicator values. Note that using parsnip::fit() will *always* create dummy variables so, if there is interest in keeping the categorical predictors in their original format, parsnip::fit_xy() would be a better choice. When using the tune package, using a recipe for pre-processing enables more control over how such predictors are encoded since recipes do not automatically create dummy variables.

Note that C5.0 has a tool for *early stopping* during boosting where less iterations of boosting are performed than the number requested. C5_rules() turns this feature off (although it can be re-enabled using C50::C5.0Control()).

Value

An updated parsnip model specification.

References

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

See Also

parsnip::fit(), parsnip::fit_xy(), C50::C5.0(), C50::C5.0Control()

Examples

```
C5_rules()
# Parameters can be represented by a placeholder:
C5_rules(trees = 7)

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

data(ad_data, package = "modeldata")

set.seed(282782)
```

```
class_rules <-
  C5_rules(trees = 1, min_n  = 10) %>%
  fit(Class ~ ., data = ad_data)

summary(class_rules$fit)

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

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

`committees` *Parameter functions for Cubist models*

Description

Committee-based models enact a boosting-like procedure to produce ensembles. `committees` parameter is for the number of models in the ensembles while `max_rules` can be used to limit the number of possible rules.

Usage

```
committees(range = c(1L, 100L), trans = NULL)

max_rules(range = c(1L, 500L), trans = NULL)
```

Arguments

`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively.
`trans`	A `trans` object from the `scales` package, such as `scales::log10_trans()` or `scales::reciprocal_trans()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Value

A function with classes "quant_param" and "param"

Examples

```
committees()
committees(4:5)

max_rules()
```

cubist_rules *General Interface for Cubist Rule-Based Regression Models*

Description

`cubist_rules()` is a way to generate a *specification* of a model before fitting. The main arguments for the model are:

- `committees`: The number of sequential models included in the ensemble (similar to the number of trees in boosting).
- `neighbors`: The number of neighbors in the post-model instance-based adjustment.

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using `parsnip::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

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

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

Arguments

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

`object`	A Cubist model specification.
`parameters`	A 1-row tibble or named list with *main* parameters to update. If the individual arguments are used, these will supersede the values in parameters. Also, using engine arguments in this object will result in an error.
`fresh`	A logical for whether the arguments should be modified in-place or replaced wholesale.
`...`	Not used for `update()`.

Details

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

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

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

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

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

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

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

Note that `cubist_rules()` does not require that categorical predictors be converted to numeric indicator values. Note that using `parsnip::fit()` will *always* create dummy variables so, if there is interest in keeping the categorical predictors in their original format, `parsnip::fit_xy()` would be a better choice. When using the `tune` package, using a recipe for pre-processing enables more control over how such predictors are encoded since recipes do not automatically create dummy variables.

The only available engine is `"Cubist"`.

Value

An updated `parsnip` model specification.

References

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

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

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

See Also

`parsnip::fit()`, `parsnip::fit_xy()`, `Cubist::cubist()`, `Cubist::cubistControl()`

Examples

```
cubist_rules()
# Parameters can be represented by a placeholder:
cubist_rules(committees = 7)

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

data(car_prices, package = "modeldata")
car_rules <-
  cubist_rules(committees = 1) %>%
  fit(log10(Price) ~ ., data = car_prices)

car_rules

summary(car_rules$fit)

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

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

---

mtry_prop *Proportion of Randomly Selected Predictors*

Description

Proportion of Randomly Selected Predictors

Usage

```
mtry_prop(range = c(0.1, 1), trans = NULL)
```

Arguments

range	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively.
trans	A trans object from the scales package, such as scales::log10_trans() or scales::reciprocal_trans(). If not provided, the default is used which matches the units used in range. If no transformation, NULL.

Value

A dials with classes "quant_param" and "param". The range element of the object is always converted to a list with elements "lower" and "upper".

---

multi_predict._C5_rules

multi_predict() *methods for rule-based models*

Description

multi_predict() methods for rule-based models

Usage

```
## S3 method for class '`_C5_rules`'
multi_predict(object, new_data, type = NULL, trees = NULL, ...)

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

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

Arguments

object	An object of class model_fit
new_data	A rectangular data object, such as a data frame.
type	A single character value or NULL. Possible values are class" and "prob".
trees	An numeric vector of trees between one and 100.
...	Not currently used.
neighbors	An numeric vector of neighbors values between zero and nine.
penalty	Non-negative penalty values.

Details

For C5.0 rule-based models, the model fit may contain less boosting iterations than the number requested. Printing the object will show how many were used due to early stopping. This can be change using an option in C50::C5.0Control(). Beware that the number of iterations requested

Value

A tibble with one row for each row of `new_data`. Multiple predictions are contained in a list column called `.pred`. That column has the standard `parsnip` prediction column names as well as the column with the tuning parameter values.

`rule_fit` *General Interface for RuleFit Models*

Description

`rule_fit()` is a way to generate a *specification* of a model before fitting. The main arguments for the model are:

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

These arguments are converted to their specific names at the time that the model is fit. Other options and argument can be set using `parsnip::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

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

## S3 method for class 'rule_fit'
update(
```

```
  object,
  parameters = NULL,
  mtry = NULL,
  trees = NULL,
  min_n = NULL,
  tree_depth = NULL,
  learn_rate = NULL,
  loss_reduction = NULL,
  sample_size = NULL,
  penalty = NULL,
  fresh = FALSE,
  ...
)
```

Arguments

mode	A single character string for the type of model. Possible values for this model are "unknown", "regression", or "classification".
mtry	An number for the number (or proportion) of predictors that will be randomly sampled at each split when creating the tree models.
trees	An integer for the number of trees contained in the ensemble.
min_n	An integer for the minimum number of data points in a node that are required for the node to be split further.
tree_depth	An integer for the maximum depth of the tree (i.e. number of splits).
learn_rate	A number for the rate at which the boosting algorithm adapts from iteration-to-iteration.
loss_reduction	A number for the reduction in the loss function required to split further .
sample_size	An number for the number (or proportion) of data that is exposed to the fitting routine.
penalty	L1 regularization parameter.
object	A rule_fit model specification.
parameters	A 1-row tibble or named list with *main* parameters to update. If the individual arguments are used, these will supersede the values in parameters. Also, using engine arguments in this object will result in an error.
fresh	A logical for whether the arguments should be modified in-place or replaced wholesale.
...	Not used for update().

Details

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

For the xrf engine, the xgboost package is used to create the rule set that is then added to a glmnet model. The only available engine is "xrf".

Differences from the xrf package:

Note that, per the documentation in `?xrf`, transformations of the response variable are not supported. To use these with `rule_fit()`, we recommend using a recipe instead of the formula method.

Also, there are several configuration differences in how `xrf()` is fit between that package and the wrapper used in `rules`. Some differences in default values are:

- `trees` (xrf: 100, rules: 15)
- `max_depth` (xrf: 3, rules: 6)

These differences will create a difference in the values of the `penalty` argument that `glmnet` uses. Also, `rules` can also set `penalty` whereas `xrf` uses an internal 5-fold cross-validation to determine it (by default).

Value

An updated `parsnip` model specification.

References

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

See Also

`parsnip::fit()`, `parsnip::fit_xy()`, `xrf::xrf.formula()`

Examples

```
rule_fit()
# Parameters can be represented by a placeholder:
rule_fit(trees = 7)

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

set.seed(6907)
rule_fit_rules <-
  rule_fit(trees = 3, penalty = 0.1) %>%
  set_mode("classification") %>%
  fit(Species ~ ., data = iris)

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

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

`tidy.cubist` *Turn regression rule models into tidy tibbles*

Description

Turn regression rule models into tidy tibbles

Usage

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

## S3 method for class 'xrf'
tidy(x, penalty = NULL, unit = c("rules", "columns"), ...)
```

Arguments

`x`	A `Cubist` or `xrf` object.
`...`	Not currently used.
`penalty`	A single numeric value for the `lambda` penalty value.
`unit`	What data should be returned? For `unit = 'rules'`, each row corresponds to a rule. For `unit = 'columns'`, each row is a predictor column. The latter can be helpful when determining variable importance.

Value

The Cubist method has columns `committee`, `rule_num`, `rule`, `estimate`, and `statistics`. The latter two are nested tibbles. `estimate` contains the parameter estimates for each term in the regression model and `statistics` has statistics about the data selected by the rules and the model fit.

The `xrf` results has columns `rule_id`, `rule`, and `estimate`. The `rule_id` column has the rule identifier (e.g., "r0_21") or the feature column name when the column is added directly into the model. For multiclass models, a `class` column is included.

In each case, the `rule` column has a character string with the rule conditions. These can be converted to an R expression using `rlang::parse_expr()`.

Examples

```
library(dplyr)

data(ames, package = "modeldata")

ames <-
  ames %>%
  mutate(Sale_Price = log10(ames$Sale_Price),
         Gr_Liv_Area = log10(ames$Gr_Liv_Area))
```

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


cb_fit <-
  cubist_rules(committees = 10) %>%
  set_engine("Cubist") %>%
  fit(Sale_Price ~ Neighborhood + Longitude + Latitude + Gr_Liv_Area + Central_Air,
      data = ames)

cb_res <- tidy(cb_fit)
cb_res

cb_res$estimate[[1]]
cb_res$statistic[[1]]


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


library(recipes)

xrf_reg_mod <-
  rule_fit(trees = 10, penalty = .001) %>%
  set_engine("xrf") %>%
  set_mode("regression")

# Make dummy variables since xgboost will not
ames_rec <-
  recipe(Sale_Price ~ Neighborhood + Longitude + Latitude +
         Gr_Liv_Area + Central_Air,
         data = ames) %>%
  step_dummy(Neighborhood, Central_Air) %>%
  step_zv(all_predictors())

ames_processed <- prep(ames_rec) %>% bake(new_data = NULL)

set.seed(1)
xrf_reg_fit <-
  xrf_reg_mod %>%
  fit(Sale_Price ~ ., data = ames_processed)

xrf_rule_res <- tidy(xrf_reg_fit)
xrf_rule_res$rule[nrow(xrf_rule_res)] %>% rlang::parse_expr()

xrf_col_res <- tidy(xrf_reg_fit, unit = "columns")
xrf_col_res
```

Index