Package 'tune'

July 8, 2020

Title Tidy Tuning Tools

Version 0.1.1

Description The ability to tune models is important. 'tune' contains functions and classes to be used in conjunction with other 'tidymodels' packages for finding reasonable values of hyper-parameters in models, pre-processing methods, and post-processing steps.

License MIT + file LICENSE

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

Depends R (>= 2.10)

Imports dplyr (>= 0.8.5), rlang (>= 0.4.0), tibble (>= 2.1.3), purrr (>= 0.3.2), dials (>= 0.0.4), recipes (>= 0.1.9), utils, ggplot2, glue, cli (>= 2.0.0), crayon, yardstick, rsample, tidyr, GPfit, foreach, parsnip (>= 0.0.4), workflows (>= 0.1.0), hardhat (>= 0.1.0), lifecycle, vctrs (>= 0.3.0)

Suggests testthat, knitr, covr, kernlab, randomForest, modeldata, xml2, spelling

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

Language en-US

NeedsCompilation no

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

Maintainer Max Kuhn <max@rstudio.com>

Repository CRAN

Date/Publication 2020-07-08 18:00:02 UTC

R topics documented:

`autoplot.tune_results` *Plot tuning search results*

Description

Plot tuning search results

Usage

```
## S3 method for class 'tune_results'
autoplot(
  object,
  type = c("marginals", "parameters", "performance"),
  metric = NULL,
  width = NULL,
  ...
)
```

Arguments

`object`	A tibble of results from `tune_grid()` or `tune_bayes()`.
`type`	A single character value. Choices are `"marginals"` (for a plot of each predictor versus performance; see Details below), `"parameters"` (each parameter versus search iteration), or `"performance"` (performance versus iteration). The latter two choices are only used for `tune_bayes()`.

metric	A character vector or NULL for which metric to plot. By default, all metrics will be shown via facets.
width	A number for the width of the confidence interval bars when type = "performance". A value of zero prevents them from being shown.
...	For plots with a regular grid, this is passed to format() and is applied to a parameter used to color points. Otherwise, it is not used.

Details

When the results of tune_grid() are used with autoplot(), it tries to determine whether a *regular grid* was used.

Regular grids:

For regular grids with one or more numeric tuning parameters, the parameter with the most unique values is used on the x-axis. If there are categorical parameters, the first is used to color the geometries. All other parameters are used in column faceting.

The plot has the performance metric(s) on the y-axis. If there are multiple metrics, these are row-faceted.

If there are more than five tuning parameters, the "marginal effects" plots are used instead.

Irregular grids:

For space-filling or random grids, a *marginal* effect plot is created. A panel is made for each numeric parameter so that each parameter is on the x-axis and performance is on the y-xis. If there are multiple metrics, these are row-faceted.

A single categorical parameter is shown as colors. If there are two or more non-numeric parameters, an error is given. A similar result occurs is only non-numeric parameters are in the grid. In these cases, we suggest using collect_metrics() and ggplot() to create a plot that is appropriate for the data.

If a parameter has an associated transformation associated with it (as determined by the parameter object used to create it), the plot shows the values in the transformed units (and is labeled with the transformation type).

Parameters are labeled using the labels found in the parameter object *except* when an identifier was used (e.g. neighbors = tune("K")).

Value

A ggplot2 object.

See Also

tune_grid(), tune_bayes()

Examples

```
# For grid search:
data("example_ames_knn")

# Plot the tuning parameter values versus performance
```

```
autoplot(ames_grid_search, metric = "rmse")


# For iterative search:
# Plot the tuning parameter values versus performance
autoplot(ames_iter_search, metric = "rmse", type = "marginals")

# Plot tuning parameters versus iterations
autoplot(ames_iter_search, metric = "rmse", type = "parameters")

# Plot performance over iterations
autoplot(ames_iter_search, metric = "rmse", type = "performance")
```

`collect_predictions` *Obtain and format results produced by tuning functions*

Description

Obtain and format results produced by tuning functions

Usage

```
collect_predictions(x, summarize = FALSE, parameters = NULL)

collect_metrics(x, summarize = TRUE)
```

Arguments

`x`	The results of `tune_grid()`, `tune_bayes()`, `fit_resamples()`, or `last_fit()`. For `collect_predictions()`, the control option `save_pred = TRUE` should have been used.
`summarize`	A logical; should metrics be summarized over resamples (`TRUE`) or return the values for each individual resample. Note that, if `x` is created by `last_fit()`, `summarize` has no effect. For the other object types, the method of summarizing predictions is detailed below.
`parameters`	An optional tibble of tuning parameter values that can be used to filter the predicted values before processing. This tibble should only have columns for each tuning parameter identifier (e.g. `"my_param"` if `tune("my_param")` was used).

Value

A tibble. The column names depend on the results and the mode of the model.

For `collect_metrics()` and `collect_predictions()`, when unsummarized, there are columns for each tuning parameter (using the `id` from `tune()`, if any). `collect_metrics()` also has columns `.metric`, and `.estimator`. When the results are summarized, there are columns for `mean`, `n`, and `std_err`. When not summarized, the additional columns for the resampling identifier(s) and `.estimate`.

For collect_predictions(), there are additional columns for the resampling identifier(s), columns for the predicted values (e.g., .pred, .pred_class, etc.), and a column for the outcome(s) using the original column name(s) in the data.

collect_predictions() can summarize the various results over replicate out-of-sample predictions. For example, when using the bootstrap, each row in the original training set has multiple holdout predictions (across assessment sets). To convert these results to a format where every training set same has a single predicted value, the results are averaged over replicate predictions.

For regression cases, the numeric predictions are simply averaged. For classification models, the problem is more complex. When class probabilities are used, these are averaged and then renormalized to make sure that they add to one. If hard class predictions also exist in the data, then these are determined from the summarized probability estimates (so that they match). If only hard class predictions are in the results, then the mode is used to summarize.

Examples

```
data("example_ames_knn")
# The parameters for the model:
parameters(ames_wflow)

# Summarized over resamples
collect_metrics(ames_grid_search)

# Per-resample values
collect_metrics(ames_grid_search, summarize = FALSE)


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

library(parsnip)
library(rsample)
library(dplyr)
library(recipes)
library(tibble)

lm_mod <-linear_reg() %>% set_engine("lm")
set.seed(93599150)
car_folds <- vfold_cv(mtcars, v = 2, repeats = 3)
ctrl <- control_resamples(save_pred = TRUE)

spline_rec <-
  recipe(mpg ~ ., data = mtcars) %>%
  step_ns(disp, deg_free = tune("df"))

grid <- tibble(df = 3:6)

resampled <- tune_grid(spline_rec, lm_mod, resamples = car_folds,
                       control = ctrl, grid = grid)

collect_predictions(resampled) %>% arrange(.row)
collect_predictions(resampled, summarize = TRUE) %>% arrange(.row)
```

```
collect_predictions(resampled, summarize = TRUE, grid[1,]) %>% arrange(.row)
```

`conf_mat_resampled` *Compute average confusion matrix across resamples*

Description

For classification problems, `conf_mat_resampled()` computes a separate confusion matrix for each resample then averages the cell counts.

Usage

```
conf_mat_resampled(x, parameters = NULL, tidy = TRUE)
```

Arguments

`x`	An object with class `tune_results` that was used with a classification model that was run with `control_*(save_pred = TRUE)`.
`parameters`	A tibble with a single tuning parameter combination. Only one tuning parameter combination (if any were used) is allowed here.
`tidy`	Should the results come back in a tibble (`TRUE`) or a matrix.

Value

A tibble or matrix with the average cell count across resamples.

Examples

```
library(parsnip)
library(rsample)
library(dplyr)

data(two_class_dat, package = "modeldata")

set.seed(2393)
res <-
  logistic_reg() %>%
  set_engine("glm") %>%
  fit_resamples(Class ~ ., resamples = vfold_cv(two_class_dat, v = 3),
                control = control_resamples(save_pred = TRUE))

conf_mat_resampled(res)
conf_mat_resampled(res, tidy = FALSE)
```

`control_bayes` *Control aspects of the Bayesian search process*

Description

Control aspects of the Bayesian search process

Usage

```
control_bayes(
  verbose = FALSE,
  no_improve = 10L,
  uncertain = Inf,
  seed = sample.int(10^5, 1),
  extract = NULL,
  save_pred = FALSE,
  time_limit = NA,
  pkgs = NULL,
  save_workflow = FALSE
)
```

Arguments

`verbose`	A logical for logging results as they are generated. Despite this argument, warnings and errors are always shown. If using a dark IDE theme, some logging messages might be hard to see. If this is the case, try setting the `tidymodels.dark` option with `options(tidymodels.dark = TRUE)` to print lighter colors.
`no_improve`	The integer cutoff for the number of iterations without better results.
`uncertain`	The number of iterations with no improvement before an uncertainty sample is created where a sample with high predicted variance is chosen (i.e., in a region that has not yet been explored). The iteration counter is reset after each uncertainty sample. For example, if `uncertain = 10`, this condition is triggered every 10 samples with no improvement.
`seed`	An integer for controlling the random number stream.
`extract`	An optional function with at least one argument (or `NULL`) that can be used to retain arbitrary objects from the model fit object, recipe, or other elements of the workflow.
`save_pred`	A logical for whether the out-of-sample predictions should be saved for each model *evaluated*.
`time_limit`	A number for the minimum number of *minutes* (elapsed) that the function should execute. The elapsed time is evaluated at internal checkpoints and, if over time, the results at that time are returned (with a warning). This means that the `time_limit` is not an exact limit, but a minimum time limit.
`pkgs`	An optional character string of R package names that should be loaded (by namespace) during parallel processing.

save_workflow A logical for whether the workflow should be appended to the output as an attribute.

Details

For `extract`, this function can be used to output the model object, the recipe (if used), or some components of either or both. When evaluated, the function's sole argument has a fitted workflow If the formula method is used, the recipe element will be `NULL`.

The results of the `extract` function are added to a list column in the output called `.extracts`. Each element of this list is a tibble with tuning parameter column and a list column (also called `.extracts`) that contains the results of the function. If no extraction function is used, there is no `.extracts` column in the resulting object. See `tune_bayes()` for more specific details.

Note that for `collect_predictions()`, it is possible that each row of the original data point might be represented multiple times per tuning parameter. For example, if the bootstrap or repeated cross-validation are used, there will be multiple rows since the sample data point has been evaluated multiple times. This may cause issues when merging the predictions with the original data.

---

control_grid *Control aspects of the grid search process*

Description

Control aspects of the grid search process

Usage

```
control_grid(
  verbose = FALSE,
  allow_par = TRUE,
  extract = NULL,
  save_pred = FALSE,
  pkgs = NULL,
  save_workflow = FALSE
)

control_resamples(
  verbose = FALSE,
  allow_par = TRUE,
  extract = NULL,
  save_pred = FALSE,
  pkgs = NULL,
  save_workflow = FALSE
)
```

Arguments

verbose	A logical for logging results as they are generated. Despite this argument, warnings and errors are always shown. If using a dark IDE theme, some logging messages might be hard to see. If this is the case, try setting the `tidymodels.dark` option with `options(tidymodels.dark = TRUE)` to print lighter colors.
allow_par	A logical to allow parallel processing (if a parallel backend is registered).
extract	An optional function with at least one argument (or `NULL`) that can be used to retain arbitrary objects from the model fit object, recipe, or other elements of the workflow.
save_pred	A logical for whether the out-of-sample predictions should be saved for each model *evaluated*.
pkgs	An optional character string of R package names that should be loaded (by namespace) during parallel processing.
save_workflow	A logical for whether the workflow should be appended to the output as an attribute.

Details

For `extract`, this function can be used to output the model object, the recipe (if used), or some components of either or both. When evaluated, the function's sole argument has a fitted workflow If the formula method is used, the recipe element will be `NULL`.

The results of the `extract` function are added to a list column in the output called `.extracts`. Each element of this list is a tibble with tuning parameter column and a list column (also called `.extracts`) that contains the results of the function. If no extraction function is used, there is no `.extracts` column in the resulting object. See `tune_bayes()` for more specific details.

Note that for `collect_predictions()`, it is possible that each row of the original data point might be represented multiple times per tuning parameter. For example, if the bootstrap or repeated cross-validation are used, there will be multiple rows since the sample data point has been evaluated multiple times. This may cause issues when merging the predictions with the original data.

`control_resamples()` is an alias for `control_grid()` and is meant to be used with `fit_resamples()`.

---

coord_obs_pred *Use same scale for plots of observed vs predicted values*

Description

For regression models, `coord_obs_pred()` can be used in a ggplot to make the x- and y-axes have the same exact scale along with an aspect ratio of one.

Usage

```
coord_obs_pred(ratio = 1, xlim = NULL, ylim = NULL, expand = TRUE, clip = "on")
```

Arguments

ratio	Aspect ratio, expressed as y / x. Defaults to 1.0.
xlim, ylim	Limits for the x and y axes.
expand	Not currently used.
clip	Should drawing be clipped to the extent of the plot panel? A setting of "on" (the default) means yes, and a setting of "off" means no. In most cases, the default of "on" should not be changed, as setting clip = "off" can cause unexpected results. It allows drawing of data points anywhere on the plot, including in the plot margins. If limits are set via xlim and ylim and some data points fall outside those limits, then those data points may show up in places such as the axes, the legend, the plot title, or the plot margins.

Value

A ggproto object.

Examples

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

library(ggplot2)
p <- ggplot(solubility_test, aes(x = solubility, y = prediction)) +
  geom_abline(lty = 2) +
  geom_point(alpha = 0.5)

p

p + coord_fixed()

p + coord_obs_pred()
```

---

example_ames_knn *Example Analysis of Ames Housing Data*

Description

Example Analysis of Ames Housing Data

Details

These objects are the results of an analysis of the Ames housing data. A K-nearest neighbors model was used with a small predictor set that included natural spline transformations of the Longitude and Latitude predictors. The code used to generate these examples was:

```
library(tidymodels)
library(tune)
library(AmesHousing)

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

ames <- make_ames()

set.seed(4595)
data_split <- initial_split(ames, strata = "Sale_Price")

ames_train <- training(data_split)

set.seed(2453)
rs_splits <- vfold_cv(ames_train, strata = "Sale_Price")

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

ames_rec <-
  recipe(Sale_Price ~ ., data = ames_train) %>%
  step_log(Sale_Price, base = 10) %>%
  step_YeoJohnson(Lot_Area, Gr_Liv_Area) %>%
  step_other(Neighborhood, threshold = .1)  %>%
  step_dummy(all_nominal()) %>%
  step_zv(all_predictors()) %>%
  step_ns(Longitude, deg_free = tune("lon")) %>%
  step_ns(Latitude, deg_free = tune("lat"))

knn_model <-
  nearest_neighbor(
    mode = "regression",
    neighbors = tune("K"),
    weight_func = tune(),
    dist_power = tune()
  ) %>%
  set_engine("kknn")

ames_wflow <-
  workflow() %>%
  add_recipe(ames_rec) %>%
  add_model(knn_model)

ames_set <-
  parameters(ames_wflow) %>%
  update(K = neighbors(c(1, 50)))

set.seed(7014)
ames_grid <-
```

```
  ames_set %>%
  grid_max_entropy(size = 10)

ames_grid_search <-
  tune_grid(
    ames_wflow,
    resamples = rs_splits,
    grid = ames_grid
  )

set.seed(2082)
ames_iter_search <-
  tune_bayes(
    ames_wflow,
    resamples = rs_splits,
    param_info = ames_set,
    initial = ames_grid_search,
    iter = 15
  )
```

important note: Since the `rsample` split columns contain a reference to the same data, saving them to disk can results in large object sizes when the object is later used. In essence, R replaces all of those references with the actual data. For this reason, we saved zero-row tibbles in their place. This doesn't affect how we use these objects in examples but be advised that using some `rsample` functions on them will cause issues.

Value

`ames_wflow` A workflow object

`ames_grid_search`, `ames_iter_search`
Results of model tuning.

Examples

```
library(tune)

ames_grid_search
ames_iter_search
```

`expo_decay` *Exponential decay function*

Description

`expo_decay()` can be used to increase or decrease a function exponentially over iterations. This can be used to dynamically set parameters for acquisition functions as iterations of Bayesian optimization proceed.

Usage

```
expo_decay(iter, start_val, limit_val, slope = 1/5)
```

Arguments

iter	An integer for the current iteration number.
start_val	The number returned for the first iteration.
limit_val	The number that the process converges to over iterations.
slope	A coefficient for the exponent to control the rate of decay. The sign of the slope controls the direction of decay.

Details

Note that, when used with the acquisition functions in `tune()`, a wrapper would be required since only the first argument would be evaluated during tuning.

Value

A single numeric value.

Examples

```
library(tibble)
library(purrr)
library(ggplot2)
library(dplyr)
tibble(
  iter = 1:40,
  value =  map_dbl(
    1:40,
    expo_decay,
    start_val = .1,
    limit_val = 0,
    slope = 1 / 5
  )
) %>%
  ggplot(aes(x = iter, y =value)) + geom_path()
```

---

extract_recipe *Convenience functions to extract model or recipe*

Description

When extracting the fitted results, the workflow is easily accessible. If there is only interest in the recipe or model, these functions can be used as shortcuts

Usage

```
extract_recipe(x)

extract_model(x)
```

Arguments

x	A fitted workflow object.

Value

A fitted model or recipe. If a formula is used instead of a recipe, `extract_recipe()` returns `NULL`.

---

filter_parameters *Remove some tuning parameter results*

Description

For objects produced by the tune_*() functions, there may only be a subset of tuning parameter combinations of interest. For large data sets, it might be helpful to be able to remove some results. This function trims the `.metrics` column of unwanted results as well as columns `.predictions` and `.extracts` (if they were requested).

Usage

```
filter_parameters(x, ..., parameters = NULL)
```

Arguments

x	An object of class `tune_results` that has multiple tuning parameters.
...	Expressions that return a logical value, and are defined in terms of the tuning parameter values. If multiple expressions are included, they are combined with the & operator. Only rows for which all conditions evaluate to `TRUE` are kept.
parameters	A tibble of tuning parameter values that can be used to filter the predicted values before processing. This tibble should only have columns for tuning parameter identifiers (e.g. `"my_param"` if `tune("my_param")` was used). There can be multiple rows and one or more columns. **If used, this parameter must be named.**

Details

Removing some parameter combinations might affect the results of `autoplot()` for the object.

Value

A version of `x` where the lists columns only retain the parameter combinations in `parameters` or satisfied by the filtering logic.

Examples

```
library(dplyr)
library(tibble)

# For grid search:
data("example_ames_knn")

## ------------------------------------------------------------------------
# select all combinations using the 'rank' weighting scheme

ames_grid_search %>%
 collect_metrics()

filter_parameters(ames_grid_search, weight_func == "rank") %>%
 collect_metrics()

rank_only <- tibble::tibble(weight_func = "rank")
filter_parameters(ames_grid_search, parameters = rank_only) %>%
 collect_metrics()

## ------------------------------------------------------------------------
# Keep only the results from the numerically best combination

ames_iter_search %>%
 collect_metrics()

best_param <- select_best(ames_iter_search, metric = "rmse")
ames_iter_search %>%
 filter_parameters(parameters = best_param) %>%
 collect_metrics()
```

`finalize_model` *Splice final parameters into objects*

Description

The finalize_* functions take a list or tibble of tuning parameter values and update objects with those values.

Usage

```
finalize_model(x, parameters)

finalize_recipe(x, parameters)

finalize_workflow(x, parameters)
```

Arguments

x	A recipe, parsnip model specification, or workflow.
parameters	A list or 1-row tibble of parameter values. Note that the column names of the tibble should be the id fields attached to tune(). For example, in the Examples section below, the model has tune("K"). In this case, the parameter tibble should be "K" and not "neighbors".

Value

An updated version of x.

Examples

```
data("example_ames_knn")

library(parsnip)
knn_model <-
  nearest_neighbor(
    mode = "regression",
    neighbors = tune("K"),
    weight_func = tune(),
    dist_power = tune()
  ) %>%
  set_engine("kknn")

lowest_rmse <- select_best(ames_grid_search, metric = "rmse")
lowest_rmse

knn_model
finalize_model(knn_model, lowest_rmse)
```

---

fit_resamples *Fit multiple models via resampling*

Description

fit_resamples() computes a set of performance metrics across one or more resamples. It does not perform any tuning (see tune_grid() and tune_bayes() for that), and is instead used for fitting a single model+recipe or model+formula combination across many resamples.

Usage

```
fit_resamples(object, ...)

## S3 method for class 'model_spec'
fit_resamples(
```

```
  object,
  preprocessor,
  resamples,
  ...,
  metrics = NULL,
  control = control_resamples()
)

## S3 method for class 'workflow'
fit_resamples(
  object,
  resamples,
  ...,
  metrics = NULL,
  control = control_resamples()
)
```

Arguments

object	A parsnip model specification or a workflows::workflow(). No tuning parameters are allowed.
...	Currently unused.
preprocessor	A traditional model formula or a recipe created using recipes::recipe().
resamples	A resample rset created from an rsample function such as rsample::vfold_cv().
metrics	A yardstick::metric_set(), or NULL to compute a standard set of metrics.
control	A control_resamples() object used to fine tune the resampling process.

Performance Metrics

To use your own performance metrics, the yardstick::metric_set() function can be used to pick what should be measured for each model. If multiple metrics are desired, they can be bundled. For example, to estimate the area under the ROC curve as well as the sensitivity and specificity (under the typical probability cutoff of 0.50), the metrics argument could be given:

```
metrics = metric_set(roc_auc, sens, spec)
```

Each metric is calculated for each candidate model.

If no metric set is provided, one is created:

- For regression models, the root mean squared error and coefficient of determination are computed.
- For classification, the area under the ROC curve and overall accuracy are computed.

Note that the metrics also determine what type of predictions are estimated during tuning. For example, in a classification problem, if metrics are used that are all associated with hard class predictions, the classification probabilities are not created.

The out-of-sample estimates of these metrics are contained in a list column called `.metrics`. This tibble contains a row for each metric and columns for the value, the estimator type, and so on.

`collect_metrics()` can be used for these objects to collapse the results over the resampled (to obtain the final resampling estimates per tuning parameter combination).

Obtaining Predictions

When `control(save_preds = TRUE)`, the output tibble contains a list column called `.predictions` that has the out-of-sample predictions for each parameter combination in the grid and each fold (which can be very large).

The elements of the tibble are tibbles with columns for the tuning parameters, the row number from the original data object (`.row`), the outcome data (with the same name(s) of the original data), and any columns created by the predictions. For example, for simple regression problems, this function generates a column called `.pred` and so on. As noted above, the prediction columns that are returned are determined by the type of metric(s) requested.

This list column can be `unnested` using `tidyr::unnest()` or using the convenience function `collect_predictions()`.

Extracting Information

The `extract` control option will result in an additional function to be returned called `.extracts`. This is a list column that has tibbles containing the results of the user's function for each tuning parameter combination. This can enable returning each model and/or recipe object that is created during resampling. Note that this could result in a large return object, depending on what is returned.

The control function contains an option (`extract`) that can be used to retain any model or recipe that was created within the resamples. This argument should be a function with a single argument. The value of the argument that is given to the function in each resample is a workflow object (see `workflows::workflow()` for more information). There are two helper functions that can be used to easily pull out the recipe (if any) and/or the model: `extract_recipe()` and `extract_model()`.

As an example, if there is interest in getting each model back, one could use:

```
extract = function (x) extract_model(x)
```

Note that the function given to the `extract` argument is evaluated on every model that is *fit* (as opposed to every model that is *evaluated*). As noted above, in some cases, model predictions can be derived for sub-models so that, in these cases, not every row in the tuning parameter grid has a separate R object associated with it.

See Also

`control_resamples()`, `collect_predictions()`, `collect_metrics()`

Examples

```
library(recipes)
library(rsample)
library(parsnip)
```

```
set.seed(6735)
folds <- vfold_cv(mtcars, v = 5)

spline_rec <- recipe(mpg ~ ., data = mtcars) %>%
  step_ns(disp) %>%
  step_ns(wt)

lin_mod <- linear_reg() %>%
  set_engine("lm")

control <- control_resamples(save_pred = TRUE)

spline_res <- fit_resamples(lin_mod, spline_rec, folds, control = control)

spline_res

show_best(spline_res, metric = "rmse")
```

`last_fit` *Fit the final best model to the training set and evaluate the test set*

Description

`last_fit()` emulates the process where, after determining the best model, the final fit on the entire training set is needed and is then evaluated on the test set.

Usage

```
last_fit(object, ...)

## S3 method for class 'model_spec'
last_fit(object, preprocessor, split, ..., metrics = NULL)

## S3 method for class 'workflow'
last_fit(object, split, ..., metrics = NULL)
```

Arguments

`object`	A `parsnip` model specification or a `workflows::workflow()`. No tuning parameters are allowed.
`...`	Currently unused.
`preprocessor`	A traditional model formula or a recipe created using `recipes::recipe()`.
`split`	An `rsplit` object created from `rsample::initial_split()`.
`metrics`	A `yardstick::metric_set()`, or `NULL` to compute a standard set of metrics.

Details

This function is intended to be used after fitting a *variety of models* and the final tuning parameters (if any) have been finalized. The next step would be to fit using the entire training set and verify performance using the test data.

Value

A single row tibble that emulates the structure of `fit_resamples()`. However, a list column called `.workflow` is also attached with the fitted model (and recipe, if any) that used the training set.

Examples

```
library(recipes)
library(rsample)
library(parsnip)

set.seed(6735)
tr_te_split <- initial_split(mtcars)

spline_rec <- recipe(mpg ~ ., data = mtcars) %>%
  step_ns(disp)

lin_mod <- linear_reg() %>%
  set_engine("lm")

spline_res <- last_fit(lin_mod, spline_rec, split = tr_te_split)
spline_res

# test set results
spline_res$.metrics[[1]]

# or use a workflow

library(workflows)
spline_wfl <-
 workflow() %>%
 add_recipe(spline_rec) %>%
 add_model(lin_mod)

last_fit(spline_wfl, split = tr_te_split)
```

`prob_improve` *Acquisition function for scoring parameter combinations*

Description

These functions can be used to score candidate tuning parameter combinations as a function of their predicted mean and variation.

Usage

```
prob_improve(trade_off = 0, eps = .Machine$double.eps)

exp_improve(trade_off = 0, eps = .Machine$double.eps)

conf_bound(kappa = 0.1)
```

Arguments

trade_off	A number or function that describes the trade-off between exploitation and exploration. Smaller values favor exploitation.
eps	A small constant to avoid division by zero.
kappa	A positive number (or function) that corresponds to the multiplier of the standard deviation in a confidence bound (e.g. 1.96 in normal-theory 95 percent confidence intervals). Smaller values lean more towards exploitation.

Details

The acquisition functions often combine the mean and variance predictions from the Gaussian process model into an objective to be optimized.

For this documentation, we assume that the metric in question is better when *maximized* (e.g. accuracy, the coefficient of determination, etc).

The expected improvement of a point `x` is based on the predicted mean and variation at that point as well as the current best value (denoted here as `x_b`). The vignette linked below contains the formulas for this acquisition function. When the `trade_off` parameter is greater than zero, the acquisition function will down-play the effect of the *mean* prediction and give more weight to the variation. This has the effect of searching for new parameter combinations that are in areas that have yet to be sampled.

Note that for `exp_improve()` and `prob_improve()`, the `trade_off` value is in the units of the outcome. The functions are parameterized so that the `trade_off` value should always be non-negative.

The confidence bound function does not take into account the current best results in the data.

If a function is passed to `exp_improve()` or `prob_improve()`, the function can have multiple arguments but only the first (the current iteration number) is given to the function. In other words, the function argument should have defaults for all but the first argument. See `expo_decay()` as an example of a function.

Value

An object of class `prob_improve`, `exp_improve`, or `conf_bounds` along with an extra class of `acquisition_function`.

See Also

`tune_bayes()`, `expo_decay()`

Examples

```
prob_improve()
```

---

show_best *Investigate best tuning parameters*

Description

show_best() displays the top sub-models and their performance estimates.

Usage

```
show_best(x, metric = NULL, n = 5, ...)

select_best(x, metric = NULL, ...)

select_by_pct_loss(x, ..., metric = NULL, limit = 2)

select_by_one_std_err(x, ..., metric = NULL)
```

Arguments

x	The results of tune_grid() or tune_bayes().
metric	A character value for the metric that will be used to sort the models. (See https://tidymodels.github.io/yardstick/articles/metric-types.html for more details). Not required if a single metric exists in x. If there are multiple metric and none are given, the first in the metric set is used (and a warning is issued).
n	An integer for the number of top results/rows to return.
...	For select_by_one_std_err() and select_by_pct_loss(), this argument is passed directly to dplyr::arrange() so that the user can sort the models from *most simple to most complex*. See the examples below. At least one term is required for these two functions.
limit	The limit of loss of performance that is acceptable (in percent units). See details below.

Details

select_best() finds the tuning parameter combination with the best performance values.

select_by_one_std_err() uses the "one-standard error rule" (Breiman _el at, 1984) that selects the most simple model that is within one standard error of the numerically optimal results.

select_by_pct_loss() selects the most simple model whose loss of performance is within some acceptable limit.

For percent loss, suppose the best model has an RMSE of 0.75 and a simpler model has an RMSE of 1. The percent loss would be `(1.00 -0.75)/1.00 * 100`, or 25 percent. Note that loss will always be non-negative.

Value

A tibble with columns for the parameters. show_best() also includes columns for performance metrics.

References

Breiman, Leo; Friedman, J. H.; Olshen, R. A.; Stone, C. J. (1984). *Classification and Regression Trees.* Monterey, CA: Wadsworth.

Examples

```
data("example_ames_knn")

show_best(ames_iter_search, metric = "rmse")

select_best(ames_iter_search, metric = "rsq")

# To find the least complex model within one std error of the numerically
# optimal model, the number of nearest neighbors are sorted from the largest
# number of neighbors (the least complex class boundary) to the smallest
# (corresponding to the most complex model).

select_by_one_std_err(ames_grid_search, metric = "rmse", desc(K))

# Now find the least complex model that has no more than a 5% loss of RMSE:
select_by_pct_loss(ames_grid_search, metric = "rmse",
                   limit = 5, desc(K))
```

---

tune *A placeholder function for argument values that are to be tuned.*

Description

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

Usage

```
tune(id = "")
```

Arguments

id	A single character value that can be used to differentiate parameters that are used in multiple places but have the same name, or if the user wants a note associated with the parameter.

Value

A call object that echos the user input.

See Also

tune_grid(), tune_bayes()

Examples

```
tune()
class(tune())
tune("your name here")

# How `tune()` is used in practice:

library(parsnip)
nearest_neighbor(
  neighbors = tune("K"),
  weight_func = tune(),
  dist_power = tune()
)
```

---

tune_bayes *Bayesian optimization of model parameters.*

Description

tune_bayes() uses models to generate new candidate tuning parameter combinations based on previous results.

Usage

```
tune_bayes(object, ...)

## S3 method for class 'model_spec'
tune_bayes(
  object,
  preprocessor,
  resamples,
  ...,
  iter = 10,
  param_info = NULL,
  metrics = NULL,
  objective = exp_improve(),
  initial = 5,
  control = control_bayes()
)
```

```
## S3 method for class 'workflow'
tune_bayes(
  object,
  resamples,
  ...,
  iter = 10,
  param_info = NULL,
  metrics = NULL,
  objective = exp_improve(),
  initial = 5,
  control = control_bayes()
)
```

Arguments

object	A parsnip model specification or a workflows::workflow().
...	Not currently used.
preprocessor	A traditional model formula or a recipe created using recipes::recipe().
resamples	An rset() object.
iter	The maximum number of search iterations.
param_info	A dials::parameters() object or NULL. If none is given, a parameters set is derived from other arguments. Passing this argument can be useful when parameter ranges need to be customized.
metrics	A yardstick::metric_set() object containing information on how models will be evaluated for performance. The first metric in metrics is the one that will be optimized.
objective	A character string for what metric should be optimized or an acquisition function object.
initial	An initial set of results in a tidy format (as would result from tune_grid()) or a positive integer. It is suggested that the number of initial results be greater than the number of parameters being optimized.
control	A control object created by control_bayes()

Details

The optimization starts with a set of initial results, such as those generated by tune_grid(). If none exist, the function will create several combinations and obtain their performance estimates.

Using one of the performance estimates as the *model outcome*, a Gaussian process (GP) model is created where the previous tuning parameter combinations are used as the predictors.

A large grid of potential hyperparameter combinations is predicted using the model and scored using an *acquisition function*. These functions usually combine the predicted mean and variance of the GP to decide the best parameter combination to try next. For more information, see the documentation for exp_improve() and the corresponding package vignette.

The best combination is evaluated using resampling and the process continues.

Value

A tibble of results that mirror those generated by `tune_grid()`. However, these results contain an `.iter` column and replicate the `rset` object multiple times over iterations (at limited additional memory costs).

Parallel Processing

The `foreach` package is used here. To execute the resampling iterations in parallel, register a parallel backend function. See the documentation for `foreach::foreach()` for examples.

For the most part, warnings generated during training are shown as they occur and are associated with a specific resample when `control(verbose = TRUE)`. They are (usually) not aggregated until the end of processing.

For Bayesian optimization, parallel processing is used to estimate the resampled performance values once a new candidate set of values are estimated.

Initial Values

The results of `tune_grid()`, or a previous run of `tune_bayes()` can be used in the `initial` argument. `initial` can also be a positive integer. In this case, a space-filling design will be used to populate a preliminary set of results. For good results, the number of initial values should be more than the number of parameters being optimized.

Parameter Ranges and Values

In some cases, the tuning parameter values depend on the dimensions of the data (they are said to contain unknown values). For example, `mtry` in random forest models depends on the number of predictors. In such cases, the unknowns in the tuning parameter object must be determined beforehand and passed to the function via the `param_info` argument. `dials::finalize()` can be used to derive the data-dependent parameters. Otherwise, a parameter set can be created via `dials::parameters()`, and the `dials` `update()` function can be used to specify the ranges or values.

Performance Metrics

To use your own performance metrics, the `yardstick::metric_set()` function can be used to pick what should be measured for each model. If multiple metrics are desired, they can be bundled. For example, to estimate the area under the ROC curve as well as the sensitivity and specificity (under the typical probability cutoff of 0.50), the `metrics` argument could be given:

```
metrics = metric_set(roc_auc, sens, spec)
```

Each metric is calculated for each candidate model.

If no metric set is provided, one is created:

- For regression models, the root mean squared error and coefficient of determination are computed.
- For classification, the area under the ROC curve and overall accuracy are computed.

Note that the metrics also determine what type of predictions are estimated during tuning. For example, in a classification problem, if metrics are used that are all associated with hard class predictions, the classification probabilities are not created.

The out-of-sample estimates of these metrics are contained in a list column called `.metrics`. This tibble contains a row for each metric and columns for the value, the estimator type, and so on.

`collect_metrics()` can be used for these objects to collapse the results over the resampled (to obtain the final resampling estimates per tuning parameter combination).

Obtaining Predictions

When `control(save_preds = TRUE)`, the output tibble contains a list column called `.predictions` that has the out-of-sample predictions for each parameter combination in the grid and each fold (which can be very large).

The elements of the tibble are tibbles with columns for the tuning parameters, the row number from the original data object (`.row`), the outcome data (with the same name(s) of the original data), and any columns created by the predictions. For example, for simple regression problems, this function generates a column called `.pred` and so on. As noted above, the prediction columns that are returned are determined by the type of metric(s) requested.

This list column can be `unnested` using `tidyr::unnest()` or using the convenience function `collect_predictions()`.

Extracting Information

The `extract` control option will result in an additional function to be returned called `.extracts`. This is a list column that has tibbles containing the results of the user's function for each tuning parameter combination. This can enable returning each model and/or recipe object that is created during resampling. Note that this could result in a large return object, depending on what is returned.

The control function contains an option (`extract`) that can be used to retain any model or recipe that was created within the resamples. This argument should be a function with a single argument. The value of the argument that is given to the function in each resample is a workflow object (see `workflows::workflow()` for more information). There are two helper functions that can be used to easily pull out the recipe (if any) and/or the model: `extract_recipe()` and `extract_model()`.

As an example, if there is interest in getting each model back, one could use:

```
extract = function (x) extract_model(x)
```

Note that the function given to the `extract` argument is evaluated on every model that is *fit* (as opposed to every model that is *evaluated*). As noted above, in some cases, model predictions can be derived for sub-models so that, in these cases, not every row in the tuning parameter grid has a separate R object associated with it.

See Also

`control_bayes()`, `tune()`, `autoplot.tune_results()`, `show_best()`, `select_best()`, `collect_predictions()`, `collect_metrics()`, `prob_improve()`, `exp_improve()`, `conf_bound()`, `fit_resamples()`

tune_grid *Model tuning via grid search*

Description

tune_grid() computes a set of performance metrics (e.g. accuracy or RMSE) for a pre-defined set of tuning parameters that correspond to a model or recipe across one or more resamples of the data.

Usage

```
tune_grid(object, ...)

## S3 method for class 'model_spec'
tune_grid(
  object,
  preprocessor,
  resamples,
  ...,
  param_info = NULL,
  grid = 10,
  metrics = NULL,
  control = control_grid()
)

## S3 method for class 'workflow'
tune_grid(
  object,
  resamples,
  ...,
  param_info = NULL,
  grid = 10,
  metrics = NULL,
  control = control_grid()
)
```

Arguments

object	A parsnip model specification or a workflows::workflow().
...	Not currently used.
preprocessor	A traditional model formula or a recipe created using recipes::recipe().
resamples	An rset() object.
param_info	A dials::parameters() object or NULL. If none is given, a parameters set is derived from other arguments. Passing this argument can be useful when parameter ranges need to be customized.

grid	A data frame of tuning combinations or a positive integer. The data frame should have columns for each parameter being tuned and rows for tuning parameter candidates. An integer denotes the number of candidate parameter sets to be created automatically.
metrics	A yardstick::metric_set() or NULL.
control	An object used to modify the tuning process.

Details

Suppose there are *m* tuning parameter combinations. tune_grid() may not require all *m* model/recipe fits across each resample. For example:

- In cases where a single model fit can be used to make predictions for different parameter values in the grid, only one fit is used. For example, for some boosted trees, if 100 iterations of boosting are requested, the model object for 100 iterations can be used to make predictions on iterations less than 100 (if all other parameters are equal).
- When the model is being tuned in conjunction with pre-processing and/or post-processing parameters, the minimum number of fits are used. For example, if the number of PCA components in a recipe step are being tuned over three values (along with model tuning parameters), only three recipes are are trained. The alternative would be to re-train the same recipe multiple times for each model tuning parameter.

The foreach package is used here. To execute the resampling iterations in parallel, register a parallel backend function. See the documentation for foreach::foreach() for examples.

For the most part, warnings generated during training are shown as they occur and are associated with a specific resample when control(verbose = TRUE). They are (usually) not aggregated until the end of processing.

Value

An updated version of resamples with extra list columns for .metrics and .notes (optional columns are .predictions and .extracts). .notes contains warnings and errors that occur during execution.

Parameter Grids

If no tuning grid is provided, a semi-random grid (via dials::grid_latin_hypercube()) is created with 10 candidate parameter combinations.

When provided, the grid should have column names for each parameter and these should be named by the parameter name or id. For example, if a parameter is marked for optimization using penalty = tune(), there should be a column names tune. If the optional identifier is used, such as penalty = tune(id = 'lambda'), then the corresponding column name should be lambda.

In some cases, the tuning parameter values depend on the dimensions of the data. For example, mtry in random forest models depends on the number of predictors. In this case, the default tuning parameter object requires an upper range. dials::finalize() can be used to derive the data-dependent parameters. Otherwise, a parameter set can be created (via dials::parameters() and the dials update() function can be used to change the values. This updated parameter set can be passed to the function via the param_info argument.

Performance Metrics

To use your own performance metrics, the `yardstick::metric_set()` function can be used to pick what should be measured for each model. If multiple metrics are desired, they can be bundled. For example, to estimate the area under the ROC curve as well as the sensitivity and specificity (under the typical probability cutoff of 0.50), the `metrics` argument could be given:

```
metrics = metric_set(roc_auc, sens, spec)
```

Each metric is calculated for each candidate model.

If no metric set is provided, one is created:

- For regression models, the root mean squared error and coefficient of determination are computed.
- For classification, the area under the ROC curve and overall accuracy are computed.

Note that the metrics also determine what type of predictions are estimated during tuning. For example, in a classification problem, if metrics are used that are all associated with hard class predictions, the classification probabilities are not created.

The out-of-sample estimates of these metrics are contained in a list column called `.metrics`. This tibble contains a row for each metric and columns for the value, the estimator type, and so on.

`collect_metrics()` can be used for these objects to collapse the results over the resampled (to obtain the final resampling estimates per tuning parameter combination).

Obtaining Predictions

When `control(save_preds = TRUE)`, the output tibble contains a list column called `.predictions` that has the out-of-sample predictions for each parameter combination in the grid and each fold (which can be very large).

The elements of the tibble are tibbles with columns for the tuning parameters, the row number from the original data object (`.row`), the outcome data (with the same name(s) of the original data), and any columns created by the predictions. For example, for simple regression problems, this function generates a column called `.pred` and so on. As noted above, the prediction columns that are returned are determined by the type of metric(s) requested.

This list column can be `unnested` using `tidyr::unnest()` or using the convenience function `collect_predictions()`.

Extracting Information

The `extract` control option will result in an additional function to be returned called `.extracts`. This is a list column that has tibbles containing the results of the user's function for each tuning parameter combination. This can enable returning each model and/or recipe object that is created during resampling. Note that this could result in a large return object, depending on what is returned.

The control function contains an option (`extract`) that can be used to retain any model or recipe that was created within the resamples. This argument should be a function with a single argument. The value of the argument that is given to the function in each resample is a workflow object (see `workflows::workflow()` for more information). There are two helper functions that can be used to easily pull out the recipe (if any) and/or the model: `extract_recipe()` and `extract_model()`.

As an example, if there is interest in getting each model back, one could use:

```
  extract = function (x) extract_model(x)
```

Note that the function given to the `extract` argument is evaluated on every model that is *fit* (as opposed to every model that is *evaluated*). As noted above, in some cases, model predictions can be derived for sub-models so that, in these cases, not every row in the tuning parameter grid has a separate R object associated with it.

See Also

`control_grid()`, `tune()`, `fit_resamples()`, `autoplot.tune_results()`, `show_best()`, `select_best()`, `collect_predictions()`, `collect_metrics()`

Examples

```
library(recipes)
library(rsample)
library(parsnip)
library(ggplot2)

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

set.seed(6735)
folds <- vfold_cv(mtcars, v = 5)

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

# tuning recipe parameters:

spline_rec <-
  recipe(mpg ~ ., data = mtcars) %>%
  step_ns(disp, deg_free = tune("disp")) %>%
  step_ns(wt, deg_free = tune("wt"))

lin_mod <-
  linear_reg() %>%
  set_engine("lm")

# manually create a grid
spline_grid <- expand.grid(disp = 2:5, wt = 2:5)

# Warnings will occur from making spline terms on the holdout data that are
# extrapolations.
spline_res <-
  tune_grid(lin_mod, spline_rec, resamples = folds, grid = spline_grid)
spline_res


show_best(spline_res, metric = "rmse")

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

```
# tune model parameters only (example requires the `kernlab` package)

car_rec <-
  recipe(mpg ~ ., data = mtcars) %>%
  step_normalize(all_predictors())

svm_mod <-
  svm_rbf(cost = tune(), rbf_sigma = tune()) %>%
  set_engine("kernlab") %>%
  set_mode("regression")

# Use a space-filling design with 7 points
set.seed(3254)
svm_res <- tune_grid(svm_mod, car_rec, resamples = folds, grid = 7)
svm_res

show_best(svm_res, metric = "rmse")

autoplot(svm_res, metric = "rmse") +
  scale_x_log10()
```

Index