Package 'tune'

May 8, 2026

Title Tidy Tuning Tools

Version 2.1.0

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, preprocessing methods, and post-processing steps.

License MIT + file LICENSE

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

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

Depends R (>= 4.1)

Imports cli (>= 3.3.0), dials (>= 1.4.3), dplyr (>= 1.1.0), GauPro (>= 0.2.15), generics (>= 0.1.2), ggplot2, glue (>= 1.6.2), hardhat (>= 1.4.2), lifecycle, parallel, parsnip (>= 1.5.0), purrr (>= 1.0.0), recipes (>= 1.3.2), rlang (>= 1.1.4), rsample (>= 1.3.2), tailor (>= 0.1.0), tibble (>= 3.1.0), tidyr (>= 1.2.0), tidyselect (>= 1.1.2), vctrs (>= 0.6.1), withr, workflows (>= 1.3.0), yardstick (>= 1.4.0)

Suggests C50, censored (>= 0.3.0), covr, future (>= 1.33.0), future.apply, kernlab, kknn, knitr, mgcv, mirai (>= 2.4.0), modeldata, probably, scales, spelling, splines2, survival, testthat (>= 3.0.0), xgboost, xml2

Config/Needs/website pkgdown, tidymodels, kknn, tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

Language en-US

LazyData true

RoxygenNote 7.3.3

NeedsCompilation no

Author Max Kuhn [aut, cre] (ORCID: <https://orcid.org/0000-0003-2402-136X>), Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Max Kuhn <max@posit.co>

Repository CRAN

Date/Publication 2026-04-17 14:30:02 UTC

Contents

`.stash_last_result` *Save most recent results to search path*

Description

Save most recent results to search path

Usage

```
.stash_last_result(x)
```

Arguments

x	An object.

Details

The function will assign `x` to `.Last.tune.result` and put it in the search path.

Value

NULL, invisibly.

.use_case_weights_with_yardstick

Determine if case weights should be passed on to yardstick

Description

This S3 method defines the logic for deciding when a case weight vector should be passed to yardstick metric functions and used to measure model performance. The current logic is that frequency weights (i.e. `hardhat::frequency_weights()`) are the only situation where this should occur.

Usage

```
.use_case_weights_with_yardstick(x)

## S3 method for class 'hardhat_importance_weights'
.use_case_weights_with_yardstick(x)

## S3 method for class 'hardhat_frequency_weights'
.use_case_weights_with_yardstick(x)
```

Arguments

x	A vector

Value

A single `TRUE` or `FALSE`.

Examples

```
library(parsnip)
library(dplyr)

frequency_weights(1:10) |>
  .use_case_weights_with_yardstick()

importance_weights(seq(1, 10, by = .1))|>
  .use_case_weights_with_yardstick()
```

`add_resample_weights` *Add resample weights to an rset object*

Description

This function allows you to specify custom weights for resamples. Weights are automatically normalized to sum to 1.

Usage

```
add_resample_weights(rset, weights)
```

Arguments

`rset`	An rset object from **rsample**.
`weights`	A numeric vector of weights, one per resample. Weights will be normalized.

Details

Resampling weights are useful when assessment sets (i.e., held out data) have different sizes or when you want to upweight certain resamples in the evaluation. The weights are stored as an attribute and used automatically during metric aggregation.

Value

The rset object with weights added as an attribute.

See Also

`calculate_resample_weights()`, `extract_resample_weights()`

Examples

```
library(rsample)
folds <- vfold_cv(mtcars, v = 3)
# Give equal weight to all folds
weighted_folds <- add_resample_weights(folds, c(1, 1, 1))
# Emphasize the first fold
weighted_folds <- add_resample_weights(folds, c(0.5, 0.25, 0.25))
```

augment.tune_results *Augment data with holdout predictions*

Description

For tune objects that use resampling, these augment() methods will add one or more columns for the hold-out predictions (i.e. from the assessment set(s)).

Usage

```
## S3 method for class 'tune_results'
augment(x, ..., parameters = NULL)

## S3 method for class 'resample_results'
augment(x, ...)

## S3 method for class 'last_fit'
augment(x, ...)
```

Arguments

x	An object resulting from one of the tune_*() functions, fit_resamples(), or last_fit(). The control specifications for these objects should have used the option save_pred = TRUE.
...	Not currently used.
parameters	A data frame with a single row that indicates what tuning parameters should be used to generate the predictions (for tune_*() objects only). If NULL, select_best(x) will be used with the first metric and, if applicable, the first evaluation time point, used to create x.

Details

For some resampling methods where rows may be replicated in multiple assessment sets, the prediction columns will be averages of the holdout results. Also, for these methods, it is possible that all rows of the original data do not have holdout predictions (like a single bootstrap resample). In this case, all rows are return and a warning is issued.

For objects created by last_fit(), the test set data and predictions are returned.

Unlike other augment() methods, the predicted values for regression models are in a column called .pred instead of .fitted (to be consistent with other tidymodels conventions).

For regression problems, an additional .resid column is added to the results.

Value

A data frame with one or more additional columns for model predictions.

`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,
  eval_time = NULL,
  width = NULL,
  call = rlang::current_env(),
  ...
)
```

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. Possible options are the entries in `.metric` column of `collect_metrics(object)`.
`eval_time`	A numeric vector of time points where dynamic event time metrics should be chosen (e.g. the time-dependent ROC curve, etc). The values should be consistent with the values used to create `object`.
`width`	A number for the width of the confidence interval bars when `type = "performance"`. A value of zero prevents them from being shown.
`call`	The call to be displayed in warnings or errors.
`...`	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")
```

`calculate_resample_weights` *Calculate resample weights from resample sizes*

Description

This convenience function calculates weights proportional to the number of observations in each resample's analysis set. Larger resamples get higher weights. This ensures that resamples with more data have proportionally more influence on the final aggregated metrics.

Usage

```
calculate_resample_weights(rset)
```

Arguments

`rset`	An rset object from **rsample**.

Details

This is particularly useful for time-based resamples (e.g., expanding window CV) or stratified sampling where resamples might have slightly different sizes, in which resamples are imbalanced.

Value

A numeric vector of weights proportional to resample sizes, normalized to sum to 1.

See Also

`add_resample_weights()`, `extract_resample_weights()`

Examples

```
library(rsample)
folds <- vfold_cv(mtcars, v = 3)
weights <- calculate_resample_weights(folds)
weighted_folds <- add_resample_weights(folds, weights)
```

collect_predictions *Obtain and format results produced by tuning functions*

Description

Obtain and format results produced by tuning functions

Usage

```
collect_predictions(x, ...)

## Default S3 method:
collect_predictions(x, ...)

## S3 method for class 'tune_results'
collect_predictions(x, ..., summarize = FALSE, parameters = NULL)

collect_metrics(x, ...)

## S3 method for class 'tune_results'
collect_metrics(x, ..., summarize = TRUE, type = c("long", "wide"))

collect_notes(x, ...)

## S3 method for class 'tune_results'
collect_notes(x, ...)

collect_extracts(x, ...)

## S3 method for class 'tune_results'
collect_extracts(x, ...)
```

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.
...	Not currently 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).

type — One of "long" (the default) or "wide". When type = "long", output has columns .metric and one of .estimate or mean. .estimate/mean gives the values for the .metric. When type = "wide", each metric has its own column and the n and std_err columns are removed, if they exist.

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 by default. For collect_metrics() methods that have a type argument, supplying type = "wide" will pivot the output such that each metric has its own column. 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 re-normalized 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.

With censored outcome models, the predicted survival probabilities (if any) are averaged while the static predicted event times are summarized using the median.

collect_notes() returns a tibble with columns for the resampling indicators, the location (preprocessor, model, etc.), type (error or warning), and the notes.

collect_extracts() collects objects extracted from fitted workflows via the extract argument to control functions. The function returns a tibble with columns for the resampling indicators, the location (preprocessor, model, etc.), and extracted objects.

Hyperparameters and extracted objects

When making use of submodels, tune can generate predictions and calculate metrics for multiple model .configurations using only one model fit. However, this means that if a function was supplied to a control function's extract argument, tune can only execute that extraction on the one model that was fitted. As a result, in the collect_extracts() output, tune opts to associate the extracted objects with the hyperparameter combination used to fit that one model workflow, rather than the hyperparameter combination of a submodel. In the output, this appears like a hyperparameter entry is recycled across many .config entries—this is intentional.

See https://parsnip.tidymodels.org/articles/Submodels.html to learn more about submodels.

Examples

```
data("example_ames_knn")
# The parameters for the model:
extract_parameter_set_dials(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, extract = extract_fit_engine)

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

grid <- tibble(df = 3:6)

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

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

collect_extracts(resampled)
```

---

compute_metrics *Calculate and format metrics from tuning functions*

Description

This function computes metrics from tuning results. The arguments and output formats are closely related to those from collect_metrics(), but this function additionally takes a metrics argument with a metric set for new metrics to compute. This allows for computing new performance metrics without requiring users to re-evaluate models against resamples.

Note that the control option save_pred = TRUE must have been supplied when generating x.

Usage

```
compute_metrics(x, metrics, summarize, event_level, ...)

## Default S3 method:
compute_metrics(x, metrics, summarize = TRUE, event_level = "first", ...)

## S3 method for class 'tune_results'
compute_metrics(x, metrics, ..., summarize = TRUE, event_level = "first")
```

Arguments

x	The results of a tuning function like tune_grid() or fit_resamples(), generated with the control option save_pred = TRUE.
metrics	A metric set of new metrics to compute. See the "Details" section below for more information.
summarize	A single logical value indicating whether metrics should be summarized over resamples (TRUE) or return the values for each individual resample. See collect_metrics() for more details on how metrics are summarized.
event_level	A single string containing either "first" or "second". This argument is passed on to yardstick metric functions when any type of class prediction is made, and specifies which level of the outcome is considered the "event".
...	Not currently used.

Details

Each metric in the set supplied to the metrics argument must have a metric type (usually "numeric", "class", or "prob") that matches some metric evaluated when generating x. e.g. For example, if x was generated with only hard "class" metrics, this function can't compute metrics that take in class probabilities ("prob".) By default, the tuning functions used to generate x compute metrics of all needed types.

Value

A tibble. See collect_metrics() for more details on the return value.

Examples

```
# load needed packages:
library(parsnip)
library(rsample)
```

```
library(yardstick)

# evaluate a linear regression against resamples.
# note that we pass `save_pred = TRUE`:
res <-
  fit_resamples(
    linear_reg(),
    mpg ~ cyl + hp,
    bootstraps(mtcars, 5),
    control = control_grid(save_pred = TRUE)
  )

# to return the metrics supplied to `fit_resamples()`:
collect_metrics(res)

# to compute new metrics:
compute_metrics(res, metric_set(mae))

# if `metrics` is the same as that passed to `fit_resamples()`,
# then `collect_metrics()` and `compute_metrics()` give the same
# output, though `compute_metrics()` is quite a bit slower:
all.equal(
  collect_metrics(res),
  compute_metrics(res, metric_set(rmse, rsq))
)
```

`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)`.
`...`	Currently unused, must be empty.
`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 `conf_mat` object like `yardstick::conf_mat()` (`FALSE`)?

Value

A tibble or `conf_mat` with the average cell count across resamples.

Examples

```
# example code

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,
  verbose_iter = 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,
  save_gp_scoring = FALSE,
```

```
  event_level = "first",
  parallel_over = NULL,
  backend_options = NULL,
  allow_par = TRUE,
  workflow_size = 100
)
```

Arguments

verbose	A logical for logging results (other than warnings and errors, which are always shown) as they are generated during training in a single R process. When using most parallel backends, this argument typically will not result in any logging. If using a dark IDE theme, some logging messages might be hard to see; try setting the `tidymodels.dark` option with `options(tidymodels.dark = TRUE)` to print lighter colors.
verbose_iter	A logical for logging results of the Bayesian search process. Defaults to FALSE. If using a dark IDE theme, some logging messages might be hard to see; 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. Tuning functions are sensitive to both the state of RNG set outside of tuning functions with `set.seed()` as well as the value set here. The value of the former determines RNG for the higher-level tuning process, like grid generation and setting the value of this argument if left as default. The value of this argument determines RNG state in workers for each iteration of model fitting, determined by the value of `parallel_over`.
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. Note that timing begins immediately on execution. Thus, if the `initial` argument to `tune_bayes()` is supplied as a number, the elapsed time will include the time needed to generate initialization results.
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.

save_gp_scoring
A logical to save the intermediate Gaussian process models for each iteration of the search. These are saved to `tempdir()` with names `gp_candidates_{i}.RData` where `i` is the iteration. These results are deleted when the R session ends. This option is only useful for teaching purposes.

event_level A single string containing either `"first"` or `"second"`. This argument is passed on to yardstick metric functions when any type of class prediction is made, and specifies which level of the outcome is considered the "event".

parallel_over A single string containing either `"resamples"` or `"everything"` describing how to use parallel processing. Alternatively, `NULL` is allowed, which chooses between `"resamples"` and `"everything"` automatically.

If `"resamples"`, then tuning will be performed in parallel over resamples alone. Within each resample, the preprocessor (i.e. recipe or formula) is processed once, and is then reused across all models that need to be fit.

If `"everything"`, then tuning will be performed in parallel at two levels. An outer parallel loop will iterate over resamples. Additionally, an inner parallel loop will iterate over all unique combinations of preprocessor and model tuning parameters for that specific resample. This will result in the preprocessor being re-processed multiple times, but can be faster if that processing is extremely fast.

If `NULL`, chooses `"resamples"` if there are more than one resample, otherwise chooses `"everything"` to attempt to maximize core utilization.

Note that switching between `parallel_over` strategies is not guaranteed to use the same random number generation schemes. However, re-tuning a model using the same `parallel_over` strategy is guaranteed to be reproducible between runs.

backend_options
An object of class `"tune_backend_options"` as created by `tune::new_backend_options()`, used to pass arguments to specific tuning backend. Defaults to `NULL` for default backend options.

allow_par A logical to allow parallel processing (if a parallel backend is registered).

workflow_size A non-negative number (in MB) that is used as a threshold for a warning regarding the size of the workflow. Only used when `save_workflow = TRUE`.

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.

Hyperparameters and extracted objects

When making use of submodels, tune can generate predictions and calculate metrics for multiple model `.configurations` using only one model fit. However, this means that if a function was supplied to a control function's `extract` argument, tune can only execute that extraction on the one model that was fitted. As a result, in the `collect_extracts()` output, tune opts to associate the extracted objects with the hyperparameter combination used to fit that one model workflow, rather than the hyperparameter combination of a submodel. In the output, this appears like a hyperparameter entry is recycled across many `.config` entries—this is intentional.

See https://parsnip.tidymodels.org/articles/Submodels.html to learn more about submodels.

---

control_last_fit *Control aspects of the last fit process*

Description

Control aspects of the last fit process

Usage

```
control_last_fit(verbose = FALSE, event_level = "first", allow_par = FALSE)
```

Arguments

verbose	A logical for logging results (other than warnings and errors, which are always shown) as they are generated during training in a single R process. When using most parallel backends, this argument typically will not result in any logging. If using a dark IDE theme, some logging messages might be hard to see; try setting the `tidymodels.dark` option with `options(tidymodels.dark = TRUE)` to print lighter colors.
event_level	A single string containing either `"first"` or `"second"`. This argument is passed on to yardstick metric functions when any type of class prediction is made, and specifies which level of the outcome is considered the "event".
allow_par	A logical to allow parallel processing (if a parallel backend is registered).

Details

`control_last_fit()` is a wrapper around `control_resamples()` and is meant to be used with `last_fit()`.

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

```
# example code
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_spline_natural(Longitude, deg_free = tune("lon")) |>
  step_spline_natural(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 <-
  extract_parameter_set_dials(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-tune *Extract elements of* tune *objects*

Description

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

- `extract_preprocessor()` returns the formula, recipe, or variable expressions used for preprocessing.
- `extract_spec_parsnip()` returns the parsnip model specification.
- `extract_fit_parsnip()` returns the parsnip model fit object.
- `extract_fit_engine()` returns the engine specific fit embedded within a parsnip model fit. For example, when using `parsnip::linear_reg()` with the `"lm"` engine, this returns the underlying `lm` object.
- `extract_mold()` returns the preprocessed "mold" object returned from `hardhat::mold()`. It contains information about the preprocessing, including either the prepped recipe, the formula terms object, or variable selectors.
- `extract_recipe()` returns the recipe. The `estimated` argument specifies whether the fitted or original recipe is returned.
- `extract_workflow()` returns the workflow object if the control option `save_workflow = TRUE` was used. The workflow will only have been estimated for objects produced by `last_fit()`.

Usage

```
## S3 method for class 'last_fit'
extract_workflow(x, ...)

## S3 method for class 'tune_results'
extract_workflow(x, ...)

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

## S3 method for class 'tune_results'
extract_recipe(x, ..., estimated = TRUE)

## S3 method for class 'tune_results'
extract_fit_parsnip(x, ...)
```

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

## S3 method for class 'tune_results'
extract_mold(x, ...)

## S3 method for class 'tune_results'
extract_preprocessor(x, ...)
```

Arguments

x	A tune_results object.
...	Not currently used.
estimated	A logical for whether the original (unfit) recipe or the fitted recipe should be returned.

Details

These functions supersede extract_model().

Value

The extracted value from the tune tune_results, x, as described in the description section.

Examples

```
# example code

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

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

spline_rec <- recipe(mpg ~ ., data = mtcars) |>
  step_spline_natural(disp)

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

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

extract_preprocessor(spline_res)

# The `spec` is the parsnip spec before it has been fit.
# The `fit` is the fitted parsnip model.
extract_spec_parsnip(spline_res)
extract_fit_parsnip(spline_res)
extract_fit_engine(spline_res)
```

```
# The mold is returned from `hardhat::mold()`, and contains the
# predictors, outcomes, and information about the preprocessing
# for use on new data at `predict()` time.
extract_mold(spline_res)

# A useful shortcut is to extract the fitted recipe from the workflow
extract_recipe(spline_res)

# That is identical to
identical(
  extract_mold(spline_res)$blueprint$recipe,
  extract_recipe(spline_res)
)
```

`extract_resample_weights` *Extract resample weights from rset or tuning objects*

Description

This function provides a consistent interface to access resample weights regardless of whether they were added to an rset object or are stored in `tune_results` after tuning.

Usage

```
extract_resample_weights(x)
```

Arguments

`x`	An rset object with resample weights, or a `tune_results` object.

Value

A numeric vector of resample weights, or NULL if no weights are present.

Examples

```
## Not run:
library(rsample)
folds <- vfold_cv(mtcars, v = 3)
weighted_folds <- add_resample_weights(folds, c(0.2, 0.3, 0.5))
extract_resample_weights(weighted_folds)

## End(Not run)
```

`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)

finalize_tailor(x, parameters)
```

Arguments

`x`	A recipe, **parsnip** model specification, **tailor** postprocessor, 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_best *Fit a model to the numerically optimal configuration*

Description

`fit_best()` takes the results from model tuning and fits it to the training set using tuning parameters associated with the best performance.

Usage

```
fit_best(x, ...)

## Default S3 method:
fit_best(x, ...)

## S3 method for class 'tune_results'
fit_best(
  x,
  ...,
  metric = NULL,
  eval_time = NULL,
  parameters = NULL,
  verbose = FALSE,
  add_validation_set = NULL
)
```

Arguments

x	The results of class tune_results (coming from functions such as tune_grid(), tune_bayes(), etc). The control option save_workflow = TRUE should have been used.
...	Not currently used, must be empty.
metric	A character string (or NULL) for which metric to optimize. If NULL, the first metric is used.
eval_time	A single numeric time point where dynamic event time metrics should be chosen (e.g., the time-dependent ROC curve, etc). The values should be consistent with the values used to create x. The NULL default will automatically use the first evaluation time used by x.
parameters	An optional 1-row tibble of tuning parameter settings, with a column for each tuning parameter. This tibble should have columns for each tuning parameter identifier (e.g. "my_param" if tune("my_param") was used). If NULL, this argument will be set to select_best(metric, eval_time). If not NULL, parameters overwrites the specification via metric, and eval_time.
verbose	A logical for printing logging.
add_validation_set	When the resamples embedded in x are a split into training set and validation set, should the validation set be included in the data set used to train the model? If not, only the training set is used. If NULL, the validation set is not used for resamples originating from rsample::validation_set() while it is used for resamples originating from rsample::validation_split().

Details

This function is a shortcut for the manual steps of:

```
best_param <- select_best(tune_results, metric) # or other `select_*()`
wflow <- finalize_workflow(wflow, best_param)  # or just `finalize_model()`
wflow_fit <- fit(wflow, data_set)
```

Value

A fitted workflow.

Case Weights

Some models can utilize case weights during training. tidymodels currently supports two types of case weights: importance weights (doubles) and frequency weights (integers). Frequency weights are used during model fitting and evaluation, whereas importance weights are only used during fitting.

To know if your model is capable of using case weights, create a model spec and test it using parsnip::case_weights_allowed().

To use them, you will need a numeric column in your data set that has been passed through either hardhat:: importance_weights() or hardhat::frequency_weights().

For functions such as `fit_resamples()` and the `tune_*()` functions, the model must be contained inside of a `workflows::workflow()`. To declare that case weights are used, invoke `workflows::add_case_weights()` with the corresponding (unquoted) column name.

From there, the packages will appropriately handle the weights during model fitting and (if appropriate) performance estimation.

See also

`last_fit()` is closely related to `fit_best()`. They both give you access to a workflow fitted on the training data but are situated somewhat differently in the modeling workflow. `fit_best()` picks up after a tuning function like `tune_grid()` to take you from tuning results to fitted workflow, ready for you to predict and assess further. `last_fit()` assumes you have made your choice of hyperparameters and finalized your workflow to then take you from finalized workflow to fitted workflow and further to performance assessment on the test data. While `fit_best()` gives a fitted workflow, `last_fit()` gives you the performance results. If you want the fitted workflow, you can extract it from the result of `last_fit()` via extract_workflow().

Examples

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

data(meats, package = "modeldata")
meats <- meats |> select(-water, -fat)

set.seed(1)
meat_split <- initial_split(meats)
meat_train <- training(meat_split)
meat_test  <- testing(meat_split)

set.seed(2)
meat_rs <- vfold_cv(meat_train, v = 10)

pca_rec <-
  recipe(protein ~ ., data = meat_train) |>
  step_normalize(all_numeric_predictors()) |>
  step_pca(all_numeric_predictors(), num_comp = tune())

knn_mod <- nearest_neighbor(neighbors = tune()) |> set_mode("regression")

ctrl <- control_grid(save_workflow = TRUE)

set.seed(128)
knn_pca_res <-
  tune_grid(knn_mod, pca_rec, resamples = meat_rs, grid = 10, control = ctrl)

knn_fit <- fit_best(knn_pca_res, verbose = TRUE)
predict(knn_fit, meat_test)
```

`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,
  eval_time = NULL,
  control = control_resamples()
)

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

Arguments

`object`	A `parsnip` model specification or an unfitted workflow(). No tuning parameters are allowed; if arguments have been marked with tune(), their values must be finalized.
`...`	Currently unused.
`preprocessor`	A traditional model formula or a recipe created using `recipes::recipe()`.
`resamples`	An `rset` resampling object created from an `rsample` function, such as `rsample::vfold_cv()`.
`metrics`	A `yardstick::metric_set()`, or `NULL` to compute a standard set of metrics.
`eval_time`	A numeric vector of time points where dynamic event time metrics should be computed (e.g. the time-dependent ROC curve, etc). The values must be non-negative and should probably be no greater than the largest event time in the training set (See Details below).

control A control_resamples() object used to fine tune the resampling process.

Case Weights

Some models can utilize case weights during training. tidymodels currently supports two types of case weights: importance weights (doubles) and frequency weights (integers). Frequency weights are used during model fitting and evaluation, whereas importance weights are only used during fitting.

To know if your model is capable of using case weights, create a model spec and test it using parsnip::case_weights_allowed().

To use them, you will need a numeric column in your data set that has been passed through either hardhat:: importance_weights() or hardhat::frequency_weights().

For functions such as fit_resamples() and the tune_*() functions, the model must be contained inside of a workflows::workflow(). To declare that case weights are used, invoke workflows::add_case_weights() with the corresponding (unquoted) column name.

From there, the packages will appropriately handle the weights during model fitting and (if appropriate) performance estimation.

Censored Regression Models

Three types of metrics can be used to assess the quality of censored regression models:

- static: the prediction is independent of time.
- dynamic: the prediction is a time-specific probability (e.g., survival probability) and is measured at one or more particular times.
- integrated: same as the dynamic metric but returns the integral of the different metrics from each time point.

Which metrics are chosen by the user affects how many evaluation times should be specified. For example:

```
# Needs no `eval_time` value
metric_set(concordance_survival)

# Needs at least one `eval_time`
metric_set(brier_survival)
metric_set(brier_survival, concordance_survival)

# Needs at least two eval_time` values
metric_set(brier_survival_integrated, concordance_survival)
metric_set(brier_survival_integrated, concordance_survival)
metric_set(brier_survival_integrated, concordance_survival, brier_survival)
```

Values of eval_time should be less than the largest observed event time in the training data. For many non-parametric models, the results beyond the largest time corresponding to an event are constant (or NA).

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.
- For censored regression, the dynamic Brier score (`yardstick::brier_survival()`) is used.
- For quantile regression, the weighted interval score (`yardstick::weighted_interval_score()`) is used.

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_grid(save_pred = 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). Several helper functions can be used to easily pull out the preprocessing and/or model information from the workflow, such as `extract_preprocessor()` and `extract_fit_parsnip()`.

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

```
extract = function (x) extract_fit_parsnip(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.

Finally, it is a good idea to include calls to `require()` for packages that are used in the function. This helps prevent failures when using parallel processing.

See Also

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

Examples

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

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

spline_rec <- recipe(mpg ~ ., data = mtcars) |>
  step_spline_natural(disp) |>
  step_spline_natural(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")

# You can also wrap up a preprocessor and a model into a workflow, and
# supply that to `fit_resamples()` instead. Here, a workflows "variables"
# preprocessor is used, which lets you supply terms using dplyr selectors.
# The variables are used as-is, no preprocessing is done to them.
wf <- workflow() |>
```

```
  add_variables(outcomes = mpg, predictors = everything()) |>
  add_model(lin_mod)

wf_res <- fit_resamples(wf, folds)
```

`int_pctl.tune_results` *Bootstrap confidence intervals for performance metrics*

Description

Using out-of-sample predictions, the bootstrap is used to create percentile confidence intervals.

Usage

```
## S3 method for class 'tune_results'
int_pctl(
  .data,
  metrics = NULL,
  eval_time = NULL,
  times = 1001,
  parameters = NULL,
  alpha = 0.05,
  allow_par = TRUE,
  event_level = "first",
  keep_replicates = FALSE,
  ...
)
```

Arguments

`.data`	A object with class `tune_results` where the `save_pred = TRUE` option was used in the control function.
`metrics`	A `yardstick::metric_set()`. By default, it uses the same metrics as the original object.
`eval_time`	A vector of evaluation times for censored regression models. `NULL` is appropriate otherwise. If `NULL` is used with censored models, a evaluation time is selected, and a warning is issued.
`times`	The number of bootstrap samples.
`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).
`alpha`	Level of significance.
`allow_par`	A logical to allow parallel processing (if a parallel backend is registered).
`event_level`	A single string. Either `"first"` or `"second"` to specify which level of truth to consider as the "event".

keep_replicates
: A logic for saving the individual estimates from each bootstrap sample (as a list column called `.values`).

...
: Not currently used.

Details

For each model configuration (if any), this function takes bootstrap samples of the out-of-sample predicted values. For each bootstrap sample, the metrics are computed and these are used to compute confidence intervals. See `rsample::int_pctl()` and the references therein for more details.

Note that the `.estimate` column is likely to be different from the results given by `collect_metrics()` since a different estimator is used. Since random numbers are used in sampling, set the random number seed prior to running this function.

The number of bootstrap samples should be large to have reliable intervals. The defaults reflect the fewest samples that should be used.

The computations for each configuration can be extensive. To increase computational efficiency parallel processing can be used. The **future** package is used here. To execute the resampling iterations in parallel, specify a plan with future first. The `allow_par` argument can be used to avoid parallelism.

Also, if a censored regression model used numerous evaluation times, the computations can take a long time unless the times are filtered with the `eval_time` argument.

Value

A tibble of metrics with additional columns for `.lower` and `.upper` (and potentially, `.values`).

References

Davison, A., & Hinkley, D. (1997). *Bootstrap Methods and their Application*. Cambridge: Cambridge University Press. doi:10.1017/CBO9780511802843

See Also

`rsample::int_pctl()`

Examples

```
if (rlang::is_installed("modeldata")) {
  data(Sacramento, package = "modeldata")
  library(rsample)
  library(parsnip)

  set.seed(13)
  sac_rs <- vfold_cv(Sacramento)

  lm_res <-
    linear_reg() |>
    fit_resamples(
      log10(price) ~ beds + baths + sqft + type + latitude + longitude,
```

```
    resamples = sac_rs,
    control = control_resamples(save_pred = TRUE)
  )

set.seed(31)
int_pctl(lm_res)
}
```

`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,
  eval_time = NULL,
  control = control_last_fit(),
  add_validation_set = FALSE
)

## S3 method for class 'workflow'
last_fit(
  object,
  split,
  ...,
  metrics = NULL,
  eval_time = NULL,
  control = control_last_fit(),
  add_validation_set = FALSE
)
```

Arguments

`object`	A `parsnip` model specification or an unfitted workflow(). No tuning parameters are allowed; if arguments have been marked with tune(), their values must be finalized.

...	Currently unused.
preprocessor	A traditional model formula or a recipe created using recipes::recipe().
split	An rsplit object created from rsample::initial_split() or rsample::initial_validation_split
metrics	A yardstick::metric_set(), or NULL to compute a standard set of metrics.
eval_time	A numeric vector of time points where dynamic event time metrics should be computed (e.g. the time-dependent ROC curve, etc). The values must be non-negative and should probably be no greater than the largest event time in the training set (See Details below).
control	A control_last_fit() object used to fine tune the last fit process.
add_validation_set	For 3-way splits into training, validation, and test set via rsample::initial_validation_split(), should the validation set be included in the data set used to train the model. If not, only the training set is used.

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. Helper functions for formatting tuning results like collect_metrics() and collect_predictions() can be used with last_fit() output.

Case Weights

Some models can utilize case weights during training. tidymodels currently supports two types of case weights: importance weights (doubles) and frequency weights (integers). Frequency weights are used during model fitting and evaluation, whereas importance weights are only used during fitting.

To know if your model is capable of using case weights, create a model spec and test it using parsnip::case_weights_allowed().

To use them, you will need a numeric column in your data set that has been passed through either hardhat:: importance_weights() or hardhat::frequency_weights().

For functions such as fit_resamples() and the tune_*() functions, the model must be contained inside of a workflows::workflow(). To declare that case weights are used, invoke workflows::add_case_weights() with the corresponding (unquoted) column name.

From there, the packages will appropriately handle the weights during model fitting and (if appropriate) performance estimation.

Censored Regression Models

Three types of metrics can be used to assess the quality of censored regression models:

- static: the prediction is independent of time.
- dynamic: the prediction is a time-specific probability (e.g., survival probability) and is measured at one or more particular times.
- integrated: same as the dynamic metric but returns the integral of the different metrics from each time point.

Which metrics are chosen by the user affects how many evaluation times should be specified. For example:

```
# Needs no `eval_time` value
metric_set(concordance_survival)

# Needs at least one `eval_time`
metric_set(brier_survival)
metric_set(brier_survival, concordance_survival)

# Needs at least two eval_time` values
metric_set(brier_survival_integrated, concordance_survival)
metric_set(brier_survival_integrated, concordance_survival)
metric_set(brier_survival_integrated, concordance_survival, brier_survival)
```

Values of `eval_time` should be less than the largest observed event time in the training data. For many non-parametric models, the results beyond the largest time corresponding to an event are constant (or NA).

See also

`last_fit()` is closely related to `fit_best()`. They both give you access to a workflow fitted on the training data but are situated somewhat differently in the modeling workflow. `fit_best()` picks up after a tuning function like `tune_grid()` to take you from tuning results to fitted workflow, ready for you to predict and assess further. `last_fit()` assumes you have made your choice of hyperparameters and finalized your workflow to then take you from finalized workflow to fitted workflow and further to performance assessment on the test data. While `fit_best()` gives a fitted workflow, `last_fit()` gives you the performance results. If you want the fitted workflow, you can extract it from the result of `last_fit()` via extract_workflow().

Examples

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

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

spline_rec <- recipe(mpg ~ ., data = mtcars) |>
```

```
  step_spline_natural(disp)

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

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

# test set metrics
collect_metrics(spline_res)

# test set predictions
collect_predictions(spline_res)

# 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)
```

`message_wrap` *Write a message that respects the line width*

Description

Write a message that respects the line width

Usage

```
message_wrap(
  x,
  width = options()$width - 2,
  prefix = "",
  color_text = NULL,
  color_prefix = color_text
)
```

Arguments

`x`	A character string of the message text.
`width`	An integer for the width.
`prefix`	An optional string to go on the first line of the message.
`color_text, color_prefix`	A function (or `NULL`) that is used to color the text and/or prefix.

Value

The processed text is returned (invisibly) but a message is written.

Examples

```
library(cli)
Gaiman <-
  paste(
    '"Good point." Bod was pleased with himself, and glad he had thought of',
    "asking the poet for advice. Really, he thought, if you couldn't trust a",
    "poet to offer sensible advice, who could you trust?",
    collapse = ""
  )
message_wrap(Gaiman)
message_wrap(Gaiman, width = 20, prefix = "-")
message_wrap(Gaiman,
  width = 30, prefix = "-",
  color_text = cli::col_silver
)
message_wrap(Gaiman,
  width = 30, prefix = "-",
  color_text = cli::style_underline,
  color_prefix = cli::col_green
)
```

`parallelism` *Support for parallel processing in tune*

Description

tune can enable simultaneous parallel computations. Tierney (2008) defined different classes of parallel processing techniques:

- *Implicit* is when a function uses low-level tools to perform a calculation that is small in scope in parallel. Examples are using multithreaded linear algebra libraries (e.g., BLAS) or basic R vectorization functions.
- *Explicit* parallelization occurs when the user requests that some calculations should be run by generating multiple new R (sub)processes. These calculations can be more complex than those for implicit parallel processing.

For example, some decision tree libraries can implicitly parallelize their search for the optimal splitting routine using multiple threads.

Alternatively, if you are resampling a model B times, you can explicitly create B new R jobs to train B boosted trees in parallel and return their resampling results to the main R process (e.g., `fit_resamples()`).

There are two frameworks that can be used to explicitly parallel process your work in **tune**: the future package and the mirai package. Previously, you could use the foreach package, but this has been deprecated as of version 1.2.1 of tune.

By default, no parallelism is used to process models in **tune**; you have to opt-in.

Using future:

You should install the package and choose your flavor of parallelism using the plan function. This allows you to specify the number of worker processes and the specific technology to use.

For example, you can use:

```
library(future)
plan(multisession, workers = 4)
```

and work will be conducted simultaneously (unless there is an exception; see the section below).

If you had previously used **foreach**, this would replace your existing code that probably looked like:

```
library(doBackend)
registerDoBackend(cores = 4)
```

See `future::plan()` for possible options other than `multisession`.

Note that **tune** resets the *maximum* limit of memory of global variables (e.g., attached packages) to be greater than the default when the package is loaded. This value can be altered using `options(future.globals.maxSize)`.

If you want **future** to use **mirai** parallel workers, you can install and load the **future.mirai** package.

Using mirai:

To set the specific for parallel processing with **mirai**, use the `mirai::daemons()` function. The first argument, `n`, determines the number of parallel workers. Using `daemons(0)` reverts to sequential processing.

The arguments `url` and `remote` are used to set up and launch parallel processes over the network for distributed computing. See `mirai::daemons()` documentation for more details.

Reverting to sequential processing:

There are a few times when you might specify that you wish to use parallel processing, but it will revert to sequential execution:

- Many of the control functions (e.g. `control_grid()`) have an argument called `allow_par`. If this is set to `FALSE`, parallel backends will always be ignored.
- Some packages, such as **rJava** and **keras** are not compatible with explicit parallelization. If any of these packages are used, sequential processing occurs.
- If you specify fewer than two workers, or if there is only a single task, the computations will occur sequentially.

Expectations for reproducibility:

We advise that you *always* run `set.seed()` with a seed value just prior to using a function that uses (or might use) random numbers. Given this:

- You should expect to get the same results if you run that section of code repeatedly, conditional on using version 1.4.0 of tune.
- You should expect differences in results between version 1.4.0 of tune and previous versions.
- When using `last_fit()`, you should be able to get the same results as manually using `generics::fit()` and `predict()` to do the same work.

- When running with or without parallel processing (using any backend package), you should be able to achieve the same results from `fit_resamples()` and the various tuning functions.

Specific exceptions:

- For SVM classification models using the **kernlab** package, the random number generator is independent of R, and there is no argument to control it. Unfortunately, it is likely to give you different results from run-to-run.
- For some deep learning packages (e.g., **tensorflow**, **keras**, and **torch**), it is very difficult to achieve reproducible results. This is especially true when using GPUs for computations. Additionally, we have seen differences in computations (stochastic or non-random) between platforms due to the packages' use of different numerical tolerance constants across operating systems.

Handling package dependencies:

tune knows what packages are required to fit a workflow object.

When computations are run sequentially, an initial check is made to see if they are installed. This triggers the packages to be loaded but not visible in the search path.

In parallel, the required packages are fully loaded (i.e., loaded and seen in the search path), as they were previously with **foreach**, in the worker processes (but not the main R session).

References

https://www.tmwr.org/grid-search#parallel-processing

Tierney, Luke. "Implicit and explicit parallel computing in R." COMPSTAT 2008: Proceedings in Computational Statistics. Physica-Verlag HD, 2008.

`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.

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.

Usage

```
show_best(x, ...)

## Default S3 method:
show_best(x, ...)

## S3 method for class 'tune_results'
show_best(
  x,
  ...,
  metric = NULL,
  eval_time = NULL,
  n = 5,
  call = rlang::current_env()
)

select_best(x, ...)

## Default S3 method:
select_best(x, ...)

## S3 method for class 'tune_results'
select_best(x, ..., metric = NULL, eval_time = NULL)

select_by_pct_loss(x, ...)

## Default S3 method:
select_by_pct_loss(x, ...)

## S3 method for class 'tune_results'
select_by_pct_loss(x, ..., metric = NULL, eval_time = NULL, limit = 2)

select_by_one_std_err(x, ...)
```

```
## Default S3 method:
select_by_one_std_err(x, ...)

## S3 method for class 'tune_results'
select_by_one_std_err(x, ..., metric = NULL, eval_time = NULL)
```

Arguments

x	The results of tune_grid() or tune_bayes().
...	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*. That is, for a parameter p, pass the unquoted expression p if smaller values of p indicate a simpler model, or desc(p) if larger values indicate a simpler model. At least one term is required for these two functions. See the examples below.
metric	A character value for the metric that will be used to sort the models. (See https://yardstick.tidymodels.org/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).
eval_time	A single numeric time point where dynamic event time metrics should be chosen (e.g., the time-dependent ROC curve, etc). The values should be consistent with the values used to create x. The NULL default will automatically use the first evaluation time used by x.
n	An integer for the number of top results/rows to return.
call	The call to be shown in errors and warnings.
limit	The limit of loss of performance that is acceptable (in percent units). See details below.

Details

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)
)
```

---

show_notes *Display distinct errors from tune objects*

Description

Display distinct errors from tune objects

Usage

```
show_notes(x, n = 10)
```

Arguments

x	An object of class `tune_results`.
n	An integer for how many unique notes to show.

Value

Invisibly, `x`. Function is called for side-effects and printing.

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,
  eval_time = 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,
  eval_time = NULL,
  objective = exp_improve(),
  initial = 5,
  control = control_bayes()
)
```

Arguments

object	A parsnip model specification or an unfitted workflow(). No tuning parameters are allowed; if arguments have been marked with tune(), their values must be finalized.
...	Not currently used.

preprocessor	A traditional model formula or a recipe created using recipes::recipe().
resamples	An rset resampling object created from an rsample function, such as rsample::vfold_cv().
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(), or NULL to compute a standard set of metrics. The first metric in metrics is the one that will be optimized.
eval_time	A numeric vector of time points where dynamic event time metrics should be computed (e.g. the time-dependent ROC curve, etc). The values must be non-negative and should probably be no greater than the largest event time in the training set (See Details below).
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

tune supports parallel processing with the **future** package. To execute the resampling iterations in parallel, specify a plan with future first. The allow_par argument can be used to avoid parallelism.

For the most part, warnings generated during training are shown as they occur and are associated with a specific resample when control_bayes(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.

The tuning parameter combinations that were tested are called *candidates*. Each candidate has a unique `.config` value that, for the initial grid search, has the pattern `pre{num}_mod{num}_post{num}`. The numbers include a zero when that element was static. For example, a value of `pre0_mod3_post4` means no preprocessors were tuned and the model and postprocessor(s) had at least three and four candidates, respectively. The iterative part of the search uses the pattern `iter{num}`. In each case, the numbers are zero-padded to enable proper sorting.

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.
- For censored regression, the dynamic Brier score (`yardstick::brier_survival()`) is used.
- For quantile regression, the weighted interval score (`yardstick::weighted_interval_score()`) is used.

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_bayes(save_pred = 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()`.

Case Weights

Some models can utilize case weights during training. tidymodels currently supports two types of case weights: importance weights (doubles) and frequency weights (integers). Frequency weights are used during model fitting and evaluation, whereas importance weights are only used during fitting.

To know if your model is capable of using case weights, create a model spec and test it using `parsnip::case_weights_allowed()`.

To use them, you will need a numeric column in your data set that has been passed through either `hardhat:: importance_weights()` or `hardhat::frequency_weights()`.

For functions such as `fit_resamples()` and the `tune_*()` functions, the model must be contained inside of a `workflows::workflow()`. To declare that case weights are used, invoke `workflows::add_case_weights()` with the corresponding (unquoted) column name.

From there, the packages will appropriately handle the weights during model fitting and (if appropriate) performance estimation.

Censored Regression Models

Three types of metrics can be used to assess the quality of censored regression models:

- static: the prediction is independent of time.
- dynamic: the prediction is a time-specific probability (e.g., survival probability) and is measured at one or more particular times.
- integrated: same as the dynamic metric but returns the integral of the different metrics from each time point.

Which metrics are chosen by the user affects how many evaluation times should be specified. For example:

```
# Needs no `eval_time` value
metric_set(concordance_survival)

# Needs at least one `eval_time`
metric_set(brier_survival)
```

```
metric_set(brier_survival, concordance_survival)

# Needs at least two eval_time` values
metric_set(brier_survival_integrated, concordance_survival)
metric_set(brier_survival_integrated, concordance_survival)
metric_set(brier_survival_integrated, concordance_survival, brier_survival)
```

Values of `eval_time` should be less than the largest observed event time in the training data. For many non-parametric models, the results beyond the largest time corresponding to an event are constant (or `NA`).

Optimizing Censored Regression Models

With dynamic performance metrics (e.g. Brier or ROC curves), performance is calculated for every value of `eval_time` but the *first* evaluation time given by the user (e.g., `eval_time[1]`) is used to guide the optimization.

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). Several helper functions can be used to easily pull out the preprocessing and/or model information from the workflow, such as `extract_preprocessor()` and `extract_fit_parsnip()`.

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

```
extract = function (x) extract_fit_parsnip(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.

Finally, it is a good idea to include calls to `require()` for packages that are used in the function. This helps prevent failures when using parallel processing.

See Also

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

Examples

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

# define resamples and minimal recipe on mtcars
set.seed(6735)
folds <- vfold_cv(mtcars, v = 5)

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

# define an svm with parameters to tune
svm_mod <-
  svm_rbf(cost = tune(), rbf_sigma = tune()) |>
  set_engine("kernlab") |>
  set_mode("regression")

# use a space-filling design with 6 points
set.seed(3254)
svm_grid <- tune_grid(svm_mod, car_rec, folds, grid = 6)

show_best(svm_grid, metric = "rmse")

# use bayesian optimization to evaluate at 6 more points
set.seed(8241)
svm_bayes <- tune_bayes(svm_mod, car_rec, folds, initial = svm_grid, iter = 6)

# note that bayesian optimization evaluated parameterizations
# similar to those that previously decreased rmse in svm_grid
show_best(svm_bayes, metric = "rmse")

# specifying `initial` as a numeric rather than previous tuning results
# will result in `tune_bayes` initially evaluating an space-filling
# grid using `tune_grid` with `grid = initial`
set.seed(0239)
svm_init <- tune_bayes(svm_mod, car_rec, folds, initial = 6, iter = 6)

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

---

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,
  eval_time = NULL,
  control = control_grid()
)

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

Arguments

object	A parsnip model specification or an unfitted workflow(). No tuning parameters are allowed; if arguments have been marked with tune(), their values must be finalized.
...	Not currently used.
preprocessor	A traditional model formula or a recipe created using recipes::recipe().
resamples	An rset resampling object created from an rsample function, such as rsample::vfold_cv().
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 to compute a standard set of metrics.
eval_time	A numeric vector of time points where dynamic event time metrics should be computed (e.g. the time-dependent ROC curve, etc). The values must be non-

negative and should probably be no greater than the largest event time in the training set (See Details below).

`control` An object used to modify the tuning process, likely created by `control_grid()`.

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 trained. The alternative would be to re-train the same recipe multiple times for each model tuning parameter.

tune supports parallel processing with the **future** package. To execute the resampling iterations in parallel, specify a plan with future first. The `allow_par` argument can be used to avoid parallelism.

For the most part, warnings generated during training are shown as they occur and are associated with a specific resample when `control_grid(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 grid (via `dials::grid_space_filling()`) 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 named `penalty`. 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.

The rows of the grid are called tuning parameter *candidates*. Each candidate has a unique `.config` value that, for grid search, has the pattern `pre{num}_mod{num}_post{num}`. The numbers include

a zero when that element was static. For example, a value of `pre0_mod3_post4` means no preprocessors were tuned and the model and postprocessor(s) had at least three and four candidates, respectively. Also, the numbers are zero-padded to enable proper sorting.

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.
- For censored regression, the dynamic Brier score (`yardstick::brier_survival()`) is used.
- For quantile regression, the weighted interval score (`yardstick::weighted_interval_score()`) is used.

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_grid(save_pred = 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). Several helper functions can be used to easily pull out the preprocessing and/or model information from the workflow, such as `extract_preprocessor()` and `extract_fit_parsnip()`.

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

```
extract = function (x) extract_fit_parsnip(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.

Finally, it is a good idea to include calls to `require()` for packages that are used in the function. This helps prevent failures when using parallel processing.

Case Weights

Some models can utilize case weights during training. tidymodels currently supports two types of case weights: importance weights (doubles) and frequency weights (integers). Frequency weights are used during model fitting and evaluation, whereas importance weights are only used during fitting.

To know if your model is capable of using case weights, create a model spec and test it using `parsnip::case_weights_allowed()`.

To use them, you will need a numeric column in your data set that has been passed through either `hardhat:: importance_weights()` or `hardhat::frequency_weights()`.

For functions such as `fit_resamples()` and the `tune_*()` functions, the model must be contained inside of a `workflows::workflow()`. To declare that case weights are used, invoke `workflows::add_case_weights()` with the corresponding (unquoted) column name.

From there, the packages will appropriately handle the weights during model fitting and (if appropriate) performance estimation.

Censored Regression Models

Three types of metrics can be used to assess the quality of censored regression models:

- static: the prediction is independent of time.
- dynamic: the prediction is a time-specific probability (e.g., survival probability) and is measured at one or more particular times.

- integrated: same as the dynamic metric but returns the integral of the different metrics from each time point.

Which metrics are chosen by the user affects how many evaluation times should be specified. For example:

```
# Needs no `eval_time` value
metric_set(concordance_survival)

# Needs at least one `eval_time`
metric_set(brier_survival)
metric_set(brier_survival, concordance_survival)

# Needs at least two eval_time` values
metric_set(brier_survival_integrated, concordance_survival)
metric_set(brier_survival_integrated, concordance_survival)
metric_set(brier_survival_integrated, concordance_survival, brier_survival)
```

Values of `eval_time` should be less than the largest observed event time in the training data. For many non-parametric models, the results beyond the largest time corresponding to an event are constant (or `NA`).

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(workflows)
library(ggplot2)

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

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

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

# tuning recipe parameters:

spline_rec <-
  recipe(mpg ~ ., data = mtcars) |>
  step_spline_natural(disp, deg_free = tune("disp")) |>
  step_spline_natural(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()

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

# Using a variables preprocessor with a workflow

# Rather than supplying a preprocessor (like a recipe) and a model directly
# to `tune_grid()`, you can also wrap them up in a workflow and pass
# that along instead (note that this doesn't do any preprocessing to
# the variables, it passes them along as-is).
wf <- workflow() |>
  add_variables(outcomes = mpg, predictors = everything()) |>
  add_model(svm_mod)

set.seed(3254)
svm_res_wf <- tune_grid(wf, resamples = folds, grid = 7)
```

Index