Package: workflowsets (via r-universe)

October 3, 2024

Title Create a Collection of 'tidymodels' Workflows

Version 1.1.0.9000

Description A workflow is a combination of a model and preprocessors (e.g, a formula, recipe, etc.) (Kuhn and Silge (2021) <https://www.tmwr.org/>). In order to try different combinations of these, an object can be created that contains many workflows. There are functions to create workflows en masse as well as training them and visualizing the results.

License MIT + file LICENSE

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

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

Depends R (>= 3.6)

Imports cli, dplyr (>= 1.0.0), generics (>= 0.1.2), ggplot2, glue, hardhat (>= 1.2.0), lifecycle (>= 1.0.0), parsnip (>= 1.2.1), pillar (>= 1.7.0), prettyunits, purrr, rlang (>= 1.1.0), rsample (>= 0.0.9), stats, tibble (>= 3.1.0), tidyr, tune (>= 1.2.0), vctrs, withr, workflows (>= 1.1.4)

Suggests covr, dials (>= 0.1.0), finetune, kknn, knitr, modeldata, recipes (>= 1.1.0), rmarkdown, spelling, testthat (>= 3.0.0), tidyclust, yardstick (>= 1.3.0)

VignetteBuilder knitr

Config/Needs/website discrim, rpart, mda, klaR, earth, tidymodels, tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

Language en-US

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Repository https://tidymodels.r-universe.dev

RemoteUrl https://github.com/tidymodels/workflowsets

RemoteRef HEAD

RemoteSha 38fbc887cf8e8b55013892738073e5d2affd60b5

Contents

`as_workflow_set` *Convert existing objects to a workflow set*

Description

Use existing objects to create a workflow set. A list of objects that are either simple workflows or objects that have class `"tune_results"` can be converted into a workflow set.

Usage

```
as_workflow_set(...)
```

Arguments

`...`	One or more named objects. Names should be unique and the objects should have at least one of the following classes: `workflow`, `iteration_results`, `tune_results`, `resample_results`, or `tune_race`. Each `tune_results` element should also contain the original workflow (accomplished using the `save_workflow` option in the control function).

Value

A workflow set. Note that the `option` column will not reflect the options that were used to create each object.

Note

The package supplies two pre-generated workflow sets, `two_class_set` and `chi_features_set`, and associated sets of model fits `two_class_res` and `chi_features_res`.

The `two_class_*` objects are based on a binary classification problem using the `two_class_dat` data from the modeldata package. The six models utilize either a bare formula or a basic recipe utilizing `recipes::step_YeoJohnson()` as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See `?two_class_set` for source code.

The `chi_features_*` objects are based on a regression problem using the `Chicago` data from the modeldata package. Each of the three models utilize a linear regression model specification, with three different recipes of varying complexity. The objects are meant to approximate the sequence of models built in Section 1.3 of Kuhn and Johnson (2019). See `?chi_features_set` for source code.

Examples

```
# ------------------------------------------------------------------------------
# Existing results

# Use the already worked example to show how to add tuned
# objects to a workflow set
two_class_res

results <- two_class_res %>% purrr::pluck("result")
names(results) <- two_class_res$wflow_id

# These are all objects that have been resampled or tuned:
purrr::map_chr(results, ~ class(.x)[1])

# Use rlang's !!! operator to splice in the elements of the list
new_set <- as_workflow_set(!!!results)

# ------------------------------------------------------------------------------
# Make a set from unfit workflows

library(parsnip)
library(workflows)

lr_spec <- logistic_reg()

main_effects <-
  workflow() %>%
  add_model(lr_spec) %>%
  add_formula(Class ~ .)

interactions <-
```

```
  workflow() %>%
  add_model(lr_spec) %>%
  add_formula(Class ~ (.)^2)

as_workflow_set(main = main_effects, int = interactions)
```

`autoplot.workflow_set` *Plot the results of a workflow set*

Description

This `autoplot()` method plots performance metrics that have been ranked using a metric. It can also run `autoplot()` on the individual results (per `wflow_id`).

Usage

```
## S3 method for class 'workflow_set'
autoplot(
  object,
  rank_metric = NULL,
  metric = NULL,
  id = "workflow_set",
  select_best = FALSE,
  std_errs = qnorm(0.95),
  type = "class",
  ...
)
```

Arguments

`object`	A `workflow_set` whose elements have results.
`rank_metric`	A character string for which metric should be used to rank the results. If none is given, the first metric in the metric set is used (after filtering by the `metric` option).
`metric`	A character vector for which metrics (apart from `rank_metric`) to be included in the visualization.
`id`	A character string for what to plot. If a value of `"workflow_set"` is used, the results of each model (and sub-model) are ordered and plotted. Alternatively, a value of the workflow set's `wflow_id` can be given and the `autoplot()` method is executed on that workflow's results.
`select_best`	A logical; should the results only contain the numerically best submodel per workflow?
`std_errs`	The number of standard errors to plot (if the standard error exists).
`type`	The aesthetics with which to differentiate workflows. The default `"class"` maps color to the model type and shape to the preprocessor type. The `"workflow"` option maps a color to each `"wflow_id"`. This argument is ignored for values of `id` other than `"workflow_set"`.
`...`	Other options to pass to `autoplot()`.

Details

This function is intended to produce a default plot to visualize helpful information across all possible applications of a workflow set. A more appropriate plot for your specific analysis can be created by calling `rank_results()` and using standard `ggplot2` code for plotting.

The x-axis is the workflow rank in the set (a value of one being the best) versus the performance metric(s) on the y-axis. With multiple metrics, there will be facets for each metric.

If multiple resamples are used, confidence bounds are shown for each result (90% confidence, by default).

Value

A ggplot object.

Note

The package supplies two pre-generated workflow sets, `two_class_set` and `chi_features_set`, and associated sets of model fits `two_class_res` and `chi_features_res`.

The `two_class_*` objects are based on a binary classification problem using the `two_class_dat` data from the modeldata package. The six models utilize either a bare formula or a basic recipe utilizing `recipes::step_YeoJohnson()` as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See `?two_class_set` for source code.

The `chi_features_*` objects are based on a regression problem using the `Chicago` data from the modeldata package. Each of the three models utilize a linear regression model specification, with three different recipes of varying complexity. The objects are meant to approximate the sequence of models built in Section 1.3 of Kuhn and Johnson (2019). See `?chi_features_set` for source code.

Examples

```
autoplot(two_class_res)
autoplot(two_class_res, select_best = TRUE)
autoplot(two_class_res, id = "yj_trans_cart", metric = "roc_auc")
```

---

chi_features_set *Chicago Features Example Data*

Description

The package supplies two pre-generated workflow sets, `two_class_set` and `chi_features_set`, and associated sets of model fits `two_class_res` and `chi_features_res`.

The `two_class_*` objects are based on a binary classification problem using the `two_class_dat` data from the modeldata package. The six models utilize either a bare formula or a basic recipe utilizing `recipes::step_YeoJohnson()` as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See `?two_class_set` for source code.

The chi_features_* objects are based on a regression problem using the Chicago data from the modeldata package. Each of the three models utilize a linear regression model specification, with three different recipes of varying complexity. The objects are meant to approximate the sequence of models built in Section 1.3 of Kuhn and Johnson (2019). See ?chi_features_set for source code.

Details

See below for the source code to generate the Chicago Features example workflow sets:

```
library(workflowsets)
library(workflows)
library(modeldata)
library(recipes)
library(parsnip)
library(dplyr)
library(rsample)
library(tune)
library(yardstick)
library(dials)

# ------------------------------------------------------------------------------
# Slightly smaller data size
data(Chicago)
Chicago <- Chicago[1:1195,]

time_val_split <-
   sliding_period(
      Chicago,
      date,
      "month",
      lookback = 38,
      assess_stop = 1
   )

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

base_recipe <-
   recipe(ridership ~ ., data = Chicago) %>%
   # create date features
   step_date(date) %>%
   step_holiday(date) %>%
   # remove date from the list of predictors
   update_role(date, new_role = "id") %>%
   # create dummy variables from factor columns
   step_dummy(all_nominal()) %>%
   # remove any columns with a single unique value
   step_zv(all_predictors()) %>%
   step_normalize(all_predictors())
```

```
date_only <-
   recipe(ridership ~ ., data = Chicago) %>%
   # create date features
   step_date(date) %>%
   update_role(date, new_role = "id") %>%
   # create dummy variables from factor columns
   step_dummy(all_nominal()) %>%
   # remove any columns with a single unique value
   step_zv(all_predictors())

date_and_holidays <-
   recipe(ridership ~ ., data = Chicago) %>%
   # create date features
   step_date(date) %>%
   step_holiday(date) %>%
   # remove date from the list of predictors
   update_role(date, new_role = "id") %>%
   # create dummy variables from factor columns
   step_dummy(all_nominal()) %>%
   # remove any columns with a single unique value
   step_zv(all_predictors())

date_and_holidays_and_pca <-
   recipe(ridership ~ ., data = Chicago) %>%
   # create date features
   step_date(date) %>%
   step_holiday(date) %>%
   # remove date from the list of predictors
   update_role(date, new_role = "id") %>%
   # create dummy variables from factor columns
   step_dummy(all_nominal()) %>%
   # remove any columns with a single unique value
   step_zv(all_predictors()) %>%
   step_pca(!!stations, num_comp = tune())

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

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

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

pca_param <-
   parameters(num_comp()) %>%
   update(num_comp = num_comp(c(0, 20)))

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

```
chi_features_set <-
   workflow_set(
      preproc = list(date = date_only,
                     plus_holidays = date_and_holidays,
                     plus_pca = date_and_holidays_and_pca),
      models = list(lm = lm_spec),
      cross = TRUE
   )

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

chi_features_res <-
   chi_features_set %>%
   option_add(param_info = pca_param, id = "plus_pca_lm") %>%
   workflow_map(resamples = time_val_split, grid = 21, seed = 1, verbose = TRUE)
```

References

Max Kuhn and Kjell Johnson (2019) *Feature Engineering and Selection*, https://bookdown.org/max/FES/a-more-complex-example.html

Examples

```
data(chi_features_set)

chi_features_set
```

`collect_metrics.workflow_set` *Obtain and format results produced by tuning functions for workflow sets*

Description

Return a tibble of performance metrics for all models or submodels.

Usage

```
## S3 method for class 'workflow_set'
collect_metrics(x, ..., summarize = TRUE)

## S3 method for class 'workflow_set'
collect_predictions(
  x,
  ...,
  summarize = TRUE,
  parameters = NULL,
  select_best = FALSE,
```

```
  metric = NULL
)

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

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

Arguments

x	A workflow_set object that has been evaluated with workflow_map().
...	Not currently used.
summarize	A logical for whether the performance estimates should be summarized via the mean (over resamples) or the raw performance values (per resample) should be returned along with the resampling identifiers. When collecting predictions, these are averaged if multiple assessment sets contain the same row.
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).
select_best	A single logical for whether the numerically best results are retained. If TRUE, the parameters argument is ignored.
metric	A character string for the metric that is used for select_best.

Details

When applied to a workflow set, the metrics and predictions that are returned do not contain the actual tuning parameter columns and values (unlike when these collect functions are run on other objects). The reason is that workflow sets can contain different types of models or models with different tuning parameters.

If the columns are needed, there are two options. First, the .config column can be used to merge the tuning parameter columns into an appropriate object. Alternatively, the map() function can be used to get the metrics from the original objects (see the example below).

Value

A tibble.

Note

The package supplies two pre-generated workflow sets, two_class_set and chi_features_set, and associated sets of model fits two_class_res and chi_features_res.

The two_class_* objects are based on a binary classification problem using the two_class_dat data from the modeldata package. The six models utilize either a bare formula or a basic recipe utilizing recipes::step_YeoJohnson() as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See ?two_class_set for source code.

The chi_features_* objects are based on a regression problem using the Chicago data from the modeldata package. Each of the three models utilize a linear regression model specification, with three different recipes of varying complexity. The objects are meant to approximate the sequence of models built in Section 1.3 of Kuhn and Johnson (2019). See ?chi_features_set for source code.

See Also

tune::collect_metrics(), rank_results()

Examples

```
library(dplyr)
library(purrr)
library(tidyr)

two_class_res

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

collect_metrics(two_class_res)

# Alternatively, if the tuning parameter values are needed:
two_class_res %>%
  dplyr::filter(grepl("cart", wflow_id)) %>%
  mutate(metrics = map(result, collect_metrics)) %>%
  dplyr::select(wflow_id, metrics) %>%
  tidyr::unnest(cols = metrics)


collect_metrics(two_class_res, summarize = FALSE)
```

---

comment_add *Add annotations and comments for workflows*

Description

comment_add() can be used to log important information about the workflow or its results as you work. Comments can be appended or removed.

Usage

```
comment_add(x, id, ..., append = TRUE, collapse = "\n")

comment_get(x, id)

comment_reset(x, id)

comment_print(x, id = NULL, ...)
```

Arguments

x	A workflow set outputted by workflow_set() or workflow_map().
id	A single character string for a value in the wflow_id column. For comment_print(), id can be a vector or NULL (and this indicates that all comments are printed).
...	One or more character strings.
append	A logical value to determine if the new comment should be added to the existing values.
collapse	A character string that separates the comments.

Value

comment_add() and comment_reset() return an updated workflow set. comment_get() returns a character string. comment_print() returns NULL invisibly.

Examples

```
two_class_set

two_class_set %>% comment_get("none_cart")

new_set <-
  two_class_set %>%
  comment_add("none_cart", "What does 'cart' stand for\u2753") %>%
  comment_add("none_cart", "Classification And Regression Trees.")

comment_print(new_set)

new_set %>% comment_get("none_cart")

new_set %>%
  comment_reset("none_cart") %>%
  comment_get("none_cart")
```

---

extract_workflow_set_result

Extract elements of workflow sets

Description

These functions extract various elements from a workflow set 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_set_result()` returns the results of `workflow_map()` for a particular workflow.
- `extract_workflow()` returns the workflow object. The workflow will not have been estimated.
- `extract_parameter_set_dials()` returns the parameter set *that will be used to fit* the supplied row `id` of the workflow set. Note that workflow sets reference a parameter set associated with the `workflow` contained in the `info` column by default, but can be fitted with a modified parameter set via the `option_add()` interface. This extractor returns the latter, if it exists, and returns the former if not, mirroring the process that `workflow_map()` follows to provide tuning functions a parameter set.
- `extract_parameter_dials()` returns the `parameters` object *that will be used to fit* the supplied tuning `parameter` in the supplied row `id` of the workflow set. See the above notes in `extract_parameter_set_dials()` on precedence.

Usage

```
extract_workflow_set_result(x, id, ...)

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

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

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

## S3 method for class 'workflow_set'
extract_fit_parsnip(x, id, ...)

## S3 method for class 'workflow_set'
extract_fit_engine(x, id, ...)

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

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

```
## S3 method for class 'workflow_set'
extract_parameter_set_dials(x, id, ...)

## S3 method for class 'workflow_set'
extract_parameter_dials(x, id, parameter, ...)
```

Arguments

x	A workflow set outputted by `workflow_set()` or `workflow_map()`.
id	A single character string for a workflow ID.
...	Other options (not currently used).
estimated	A logical for whether the original (unfit) recipe or the fitted recipe should be returned.
parameter	A single string for the parameter ID.

Details

These functions supersede the `pull_*()` functions (e.g., `extract_workflow_set_result()`).

Value

The extracted value from the object, `x`, as described in the description section.

Note

The package supplies two pre-generated workflow sets, `two_class_set` and `chi_features_set`, and associated sets of model fits `two_class_res` and `chi_features_res`.

The `two_class_*` objects are based on a binary classification problem using the `two_class_dat` data from the modeldata package. The six models utilize either a bare formula or a basic recipe utilizing `recipes::step_YeoJohnson()` as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See `?two_class_set` for source code.

The `chi_features_*` objects are based on a regression problem using the `Chicago` data from the modeldata package. Each of the three models utilize a linear regression model specification, with three different recipes of varying complexity. The objects are meant to approximate the sequence of models built in Section 1.3 of Kuhn and Johnson (2019). See `?chi_features_set` for source code.

Examples

```
library(tune)

two_class_res

extract_workflow_set_result(two_class_res, "none_cart")

extract_workflow(two_class_res, "none_cart")
```

`fit_best.workflow_set` *Fit a model to the numerically optimal configuration*

Description

`fit_best()` takes results from tuning many models and fits the workflow configuration associated with the best performance to the training set.

Usage

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

Arguments

`x`	A `workflow_set` object that has been evaluated with `workflow_map()`. Note that the workflow set must have been fitted with the control option `save_workflow = TRUE`.
`metric`	A character string giving the metric to rank results by.
`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`.
`...`	Additional options to pass to tune::fit_best.

Details

This function is a shortcut for the steps needed to fit the numerically optimal configuration in a fitted workflow set. The function ranks results, extracts the tuning result pertaining to the best result, and then again calls `fit_best()` (itself a wrapper) on the tuning result containing the best result.

In pseudocode:

```
rankings <- rank_results(wf_set, metric, select_best = TRUE)
tune_res <- extract_workflow_set_result(wf_set, rankings$wflow_id[1])
fit_best(tune_res, metric)
```

Note

The package supplies two pre-generated workflow sets, `two_class_set` and `chi_features_set`, and associated sets of model fits `two_class_res` and `chi_features_res`.

The `two_class_*` objects are based on a binary classification problem using the `two_class_dat` data from the modeldata package. The six models utilize either a bare formula or a basic recipe utilizing `recipes::step_YeoJohnson()` as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See `?two_class_set` for source code.

The `chi_features_*` objects are based on a regression problem using the `Chicago` data from the modeldata package. Each of the three models utilize a linear regression model specification, with

three different recipes of varying complexity. The objects are meant to approximate the sequence of models built in Section 1.3 of Kuhn and Johnson (2019). See ?chi_features_set for source code.

Examples

```
library(tune)
library(modeldata)
library(rsample)

data(Chicago)
Chicago <- Chicago[1:1195,]

time_val_split <-
   sliding_period(
      Chicago,
      date,
      "month",
      lookback = 38,
      assess_stop = 1
   )

chi_features_set

chi_features_res_new <-
   chi_features_set %>%
   # note: must set `save_workflow = TRUE` to use `fit_best()`
   option_add(control = control_grid(save_workflow = TRUE)) %>%
   # evaluate with resamples
   workflow_map(resamples = time_val_split, grid = 21, seed = 1, verbose = TRUE)

chi_features_res_new

# sort models by performance metrics
rank_results(chi_features_res_new)

# fit the numerically optimal configuration to the training set
chi_features_wf <- fit_best(chi_features_res_new)

chi_features_wf

# to select optimal value based on a specific metric:
fit_best(chi_features_res_new, metric = "rmse")
```

---

leave_var_out_formulas

Create formulas without each predictor

Description

From an initial model formula, create a list of formulas that exclude each predictor.

Usage

```
leave_var_out_formulas(formula, data, full_model = TRUE, ...)
```

Arguments

formula	A model formula that contains at least two predictors.
data	A data frame.
full_model	A logical; should the list include the original formula?
...	Options to pass to stats::model.frame()

Details

The new formulas obey the hierarchy rule so that interactions without main effects are not included (unless the original formula contains such terms).

Factor predictors are left as-is (i.e., no indicator variables are created).

Value

A named list of formulas

See Also

workflow_set()

Examples

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

leave_var_out_formulas(
  bill_length_mm ~ .,
  data = penguins
)

leave_var_out_formulas(
  bill_length_mm ~ (island + sex)^2 + flipper_length_mm,
  data = penguins
)

leave_var_out_formulas(
  bill_length_mm ~ (island + sex)^2 + flipper_length_mm +
    I(flipper_length_mm^2),
  data = penguins
)
```

option_add *Add and edit options saved in a workflow set*

Description

The option column controls options for the functions that are used to *evaluate* the workflow set, such as tune::fit_resamples() or tune::tune_grid(). Examples of common options to set for these functions include param_info and grid.

These functions are helpful for manipulating the information in the option column.

Usage

```
option_add(x, ..., id = NULL, strict = FALSE)

option_remove(x, ...)

option_add_parameters(x, id = NULL, strict = FALSE)
```

Arguments

x	A workflow set outputted by workflow_set() or workflow_map().
...	Arguments to pass to the tune_*() functions (e.g. tune::tune_grid()) or tune::fit_resamples(). For option_remove() this can be a series of unquoted option names.
id	A character string of one or more values from the wflow_id column that indicates which options to update. By default, all workflows are updated.
strict	A logical; should execution stop if existing options are being replaced?

Details

option_add() is used to update all of the options in a workflow set.

option_remove() will eliminate specific options across rows.

option_add_parameters() adds a parameter object to the option column (if parameters are being tuned).

Note that executing a function on the workflow set, such as tune_grid(), will add any options given to that function to the option column.

These functions do *not* control options for the individual workflows, such as the recipe blueprint. When creating a workflow manually, use workflows::add_model() or workflows::add_recipe() to specify extra options. To alter these in a workflow set, use update_workflow_model() or update_workflow_recipe().

Value

An updated workflow set.

Examples

```
library(tune)

two_class_set

two_class_set %>%
  option_add(grid = 10)

two_class_set %>%
  option_add(grid = 10) %>%
  option_add(grid = 50, id = "none_cart")

two_class_set %>%
  option_add_parameters()
```

`option_list` *Make a classed list of options*

Description

This function returns a named list with an extra class of `"workflow_set_options"` that has corresponding formatting methods for printing inside of tibbles.

Usage

```
option_list(...)
```

Arguments

`...`	A set of named options (or nothing)

Value

A classed list.

Examples

```
option_list(a = 1, b = 2)
option_list()
```

pull_workflow_set_result *Extract elements from a workflow set*

Description

[Soft-deprecated]

Usage

```
pull_workflow_set_result(x, id)

pull_workflow(x, id)
```

Arguments

x	A workflow set outputted by workflow_set() or workflow_map().
id	A single character string for a workflow ID.

Details

pull_workflow_set_result() retrieves the results of workflow_map() for a particular workflow while pull_workflow() extracts the unfitted workflow from the info column.

The extract_workflow_set_result() and extract_workflow() functions should be used instead of these functions.

Value

pull_workflow_set_result() produces a tune_result or resample_results object. pull_workflow() returns an unfit workflow object.

Examples

```
library(tune)

two_class_res

pull_workflow_set_result(two_class_res, "none_cart")

pull_workflow(two_class_res, "none_cart")
```

rank_results *Rank the results by a metric*

Description

This function sorts the results by a specific performance metric.

Usage

```
rank_results(x, rank_metric = NULL, eval_time = NULL, select_best = FALSE)
```

Arguments

x	A workflow_set object that has been evaluated with workflow_map().
rank_metric	A character string for a metric.
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.
select_best	A logical giving whether the results should only contain the numerically best submodel per workflow.

Details

If some models have the exact same performance, rank(value, ties.method = "random") is used (with a reproducible seed) so that all ranks are integers.

No columns are returned for the tuning parameters since they are likely to be different (or not exist) for some models. The wflow_id and .config columns can be used to determine the corresponding parameter values.

Value

A tibble with columns: wflow_id, .config, .metric, mean, std_err, n, preprocessor, model, and rank.

Note

The package supplies two pre-generated workflow sets, two_class_set and chi_features_set, and associated sets of model fits two_class_res and chi_features_res.

The two_class_* objects are based on a binary classification problem using the two_class_dat data from the modeldata package. The six models utilize either a bare formula or a basic recipe utilizing recipes::step_YeoJohnson() as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See ?two_class_set for source code.

The chi_features_* objects are based on a regression problem using the Chicago data from the modeldata package. Each of the three models utilize a linear regression model specification, with three different recipes of varying complexity. The objects are meant to approximate the sequence

of models built in Section 1.3 of Kuhn and Johnson (2019). See ?chi_features_set for source code.

Examples

```
chi_features_res

rank_results(chi_features_res)
rank_results(chi_features_res, select_best = TRUE)
rank_results(chi_features_res, rank_metric = "rsq")
```

---

two_class_set *Two Class Example Data*

Description

The package supplies two pre-generated workflow sets, two_class_set and chi_features_set, and associated sets of model fits two_class_res and chi_features_res.

The two_class_* objects are based on a binary classification problem using the two_class_dat data from the modeldata package. The six models utilize either a bare formula or a basic recipe utilizing recipes::step_YeoJohnson() as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See ?two_class_set for source code.

The chi_features_* objects are based on a regression problem using the Chicago data from the modeldata package. Each of the three models utilize a linear regression model specification, with three different recipes of varying complexity. The objects are meant to approximate the sequence of models built in Section 1.3 of Kuhn and Johnson (2019). See ?chi_features_set for source code.

Details

See below for the source code to generate the Two Class example workflow sets:

```
library(workflowsets)
library(workflows)
library(modeldata)
library(recipes)
library(parsnip)
library(dplyr)
library(rsample)
library(tune)
library(yardstick)

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

data(two_class_dat, package = "modeldata")

set.seed(1)
```

```
folds <- vfold_cv(two_class_dat, v = 5)

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

decision_tree_rpart_spec <-
  decision_tree(min_n = tune(), cost_complexity = tune()) %>%
  set_engine('rpart') %>%
  set_mode('classification')

logistic_reg_glm_spec <-
  logistic_reg() %>%
  set_engine('glm')

mars_earth_spec <-
  mars(prod_degree = tune()) %>%
  set_engine('earth') %>%
  set_mode('classification')

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

yj_recipe <-
   recipe(Class ~ ., data = two_class_dat) %>%
   step_YeoJohnson(A, B)

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

two_class_set <-
   workflow_set(
      preproc = list(none = Class ~ A + B, yj_trans = yj_recipe),
     models = list(cart = decision_tree_rpart_spec, glm = logistic_reg_glm_spec,
                   mars = mars_earth_spec)
   )

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

two_class_res <-
  two_class_set %>%
  workflow_map(
    resamples = folds,
    grid = 10,
    seed = 2,
    verbose = TRUE,
    control = control_grid(save_workflow = TRUE)
  )
```

Examples

```
data(two_class_set)
```

```
two_class_set
```

`update_workflow_model` *Update components of a workflow within a workflow set*

Description

Workflows can take special arguments for the recipe (e.g. a blueprint) or a model (e.g. a special formula). However, when creating a workflow set, there is no way to specify these extra components.

`update_workflow_model()` and `update_workflow_recipe()` allow users to set these values *after* the workflow set is initially created. They are analogous to `workflows::add_model()` or `workflows::add_recipe()`.

Usage

```
update_workflow_model(x, id, spec, formula = NULL)

update_workflow_recipe(x, id, recipe, blueprint = NULL)
```

Arguments

`x`	A workflow set outputted by `workflow_set()` or `workflow_map()`.
`id`	A single character string from the `wflow_id` column indicating which workflow to update.
`spec`	A parsnip model specification.
`formula`	An optional formula override to specify the terms of the model. Typically, the terms are extracted from the formula or recipe preprocessing methods. However, some models (like survival and bayesian models) use the formula not to preprocess, but to specify the structure of the model. In those cases, a formula specifying the model structure must be passed unchanged into the model call itself. This argument is used for those purposes.
`recipe`	A recipe created using `recipes::recipe()`. The recipe should not have been trained already with `recipes::prep()`; workflows will handle training internally.
`blueprint`	A hardhat blueprint used for fine tuning the preprocessing. If `NULL`, `hardhat::default_recipe_blueprint()` is used. Note that preprocessing done here is separate from preprocessing that might be done automatically by the underlying model.

Note

The package supplies two pre-generated workflow sets, `two_class_set` and `chi_features_set`, and associated sets of model fits `two_class_res` and `chi_features_res`.

The `two_class_*` objects are based on a binary classification problem using the `two_class_dat` data from the modeldata package. The six models utilize either a bare formula or a basic recipe

utilizing `recipes::step_YeoJohnson()` as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See `?two_class_set` for source code.

The `chi_features_*` objects are based on a regression problem using the `Chicago` data from the modeldata package. Each of the three models utilize a linear regression model specification, with three different recipes of varying complexity. The objects are meant to approximate the sequence of models built in Section 1.3 of Kuhn and Johnson (2019). See `?chi_features_set` for source code.

Examples

```
library(parsnip)

new_mod <-
  decision_tree() %>%
  set_engine("rpart", method = "anova") %>%
  set_mode("classification")

new_set <- update_workflow_model(two_class_res, "none_cart", spec = new_mod)

new_set

extract_workflow(new_set, id = "none_cart")
```

---

workflow_map *Process a series of workflows*

Description

`workflow_map()` will execute the same function across the workflows in the set. The various `tune_*()` functions can be used as well as `tune::fit_resamples()`.

Usage

```
workflow_map(
  object,
  fn = "tune_grid",
  verbose = FALSE,
  seed = sample.int(10^4, 1),
  ...
)
```

Arguments

`object`	A workflow set.
`fn`	The name of the function to run, as a character. Acceptable values are: "tune_grid", "tune_bayes", "fit_resamples", "tune_race_anova", "tune_race_win_loss", or "tune_sim_anneal". Note that users need not provide the namespace or parentheses in this argument, e.g. provide `"tune_grid"` rather than `"tune::tune_grid"` or `"tune_grid()"`.

verbose	A logical for logging progress.
seed	A single integer that is set prior to each function execution.
...	Options to pass to the modeling function. See details below.

Details

When passing options, anything passed in the ... will be combined with any values in the option column. The values in ... will override that column's values and the new options are added to the options column.

Any failures in execution result in the corresponding row of results to contain a try-error object.

In cases where a model has no tuning parameters is mapped to one of the tuning functions, tune::fit_resamples() will be used instead and a warning is issued if verbose = TRUE.

If a workflow requires packages that are not installed, a message is printed and workflow_map() continues with the next workflow (if any).

Value

An updated workflow set. The option column will be updated with any options for the tune package functions given to workflow_map(). Also, the results will be added to the result column. If the computations for a workflow fail, a try-catch object will be saved in place of the results (without stopping execution).

Note

The package supplies two pre-generated workflow sets, two_class_set and chi_features_set, and associated sets of model fits two_class_res and chi_features_res.

The two_class_* objects are based on a binary classification problem using the two_class_dat data from the modeldata package. The six models utilize either a bare formula or a basic recipe utilizing recipes::step_YeoJohnson() as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See ?two_class_set for source code.

The chi_features_* objects are based on a regression problem using the Chicago data from the modeldata package. Each of the three models utilize a linear regression model specification, with three different recipes of varying complexity. The objects are meant to approximate the sequence of models built in Section 1.3 of Kuhn and Johnson (2019). See ?chi_features_set for source code.

See Also

workflow_set(), as_workflow_set(), extract_workflow_set_result()

Examples

```
library(workflowsets)
library(workflows)
library(modeldata)
library(recipes)
library(parsnip)
library(dplyr)
```

```
library(rsample)
library(tune)
library(yardstick)
library(dials)

# An example of processed results
chi_features_res

# Recreating them:

# ------------------------------------------------------------------------------
data(Chicago)
Chicago <- Chicago[1:1195,]

time_val_split <-
   sliding_period(
      Chicago,
      date,
      "month",
      lookback = 38,
      assess_stop = 1
   )

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

base_recipe <-
   recipe(ridership ~ ., data = Chicago) %>%
   # create date features
   step_date(date) %>%
   step_holiday(date) %>%
   # remove date from the list of predictors
   update_role(date, new_role = "id") %>%
   # create dummy variables from factor columns
   step_dummy(all_nominal()) %>%
   # remove any columns with a single unique value
   step_zv(all_predictors()) %>%
   step_normalize(all_predictors())

date_only <-
   recipe(ridership ~ ., data = Chicago) %>%
   # create date features
   step_date(date) %>%
   update_role(date, new_role = "id") %>%
   # create dummy variables from factor columns
   step_dummy(all_nominal()) %>%
   # remove any columns with a single unique value
   step_zv(all_predictors())

date_and_holidays <-
   recipe(ridership ~ ., data = Chicago) %>%
   # create date features
   step_date(date) %>%
   step_holiday(date) %>%
```

```
   # remove date from the list of predictors
   update_role(date, new_role = "id") %>%
   # create dummy variables from factor columns
   step_dummy(all_nominal()) %>%
   # remove any columns with a single unique value
   step_zv(all_predictors())

date_and_holidays_and_pca <-
   recipe(ridership ~ ., data = Chicago) %>%
   # create date features
   step_date(date) %>%
   step_holiday(date) %>%
   # remove date from the list of predictors
   update_role(date, new_role = "id") %>%
   # create dummy variables from factor columns
   step_dummy(all_nominal()) %>%
   # remove any columns with a single unique value
   step_zv(all_predictors()) %>%
   step_pca(!!stations, num_comp = tune())

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

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

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

pca_param <-
   parameters(num_comp()) %>%
   update(num_comp = num_comp(c(0, 20)))

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

chi_features_set <-
   workflow_set(
      preproc = list(date = date_only,
                     plus_holidays = date_and_holidays,
                     plus_pca = date_and_holidays_and_pca),
      models = list(lm = lm_spec),
      cross = TRUE
   )

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

chi_features_res_new <-
   chi_features_set %>%
   option_add(param_info = pca_param, id = "plus_pca_lm") %>%
   workflow_map(resamples = time_val_split, grid = 21, seed = 1, verbose = TRUE)

chi_features_res_new
```

`workflow_set` *Generate a set of workflow objects from preprocessing and model objects*

Description

Often a data practitioner needs to consider a large number of possible modeling approaches for a task at hand, especially for new data sets and/or when there is little knowledge about what modeling strategy will work best. Workflow sets provide an expressive interface for investigating multiple models or feature engineering strategies in such a situation.

Usage

```
workflow_set(preproc, models, cross = TRUE, case_weights = NULL)
```

Arguments

`preproc`	A list (preferably named) with preprocessing objects: formulas, recipes, or `workflows::workflow_variables()`.
`models`	A list (preferably named) of `parsnip` model specifications.
`cross`	A logical: should all combinations of the preprocessors and models be used to create the workflows? If `FALSE`, the length of `preproc` and `models` should be equal.
`case_weights`	A single unquoted column name specifying the case weights for the models. This must be a classed case weights column, as determined by `hardhat::is_case_weights()`. See the "Case weights" section below for more information.

Details

The preprocessors that can be combined with the model objects can be one or more of:

- A traditional R formula.
- A recipe definition (un-prepared) via `recipes::recipe()`.
- A selectors object created by `workflows::workflow_variables()`.

Since `preproc` is a named list column, any combination of these can be used in that argument (i.e., `preproc` can be mixed types).

Value

A tibble with extra class 'workflow_set'. A new set includes four columns (but others can be added):

- `wflow_id` contains character strings for the preprocessor/workflow combination. These can be changed but must be unique.
- `info` is a list column with tibbles containing more specific information, including any comments added using `comment_add()`. This tibble also contains the workflow object (which can be easily retrieved using `extract_workflow()`).

- option is a list column that will include a list of optional arguments passed to the functions from the tune package. They can be added manually via option_add() or automatically when options are passed to workflow_map().
- result is a list column that will contain any objects produced when workflow_map() is used.

Case weights

The case_weights argument can be passed as a single unquoted column name identifying the data column giving model case weights. For each workflow in the workflow set using an engine that supports case weights, the case weights will be added with workflows::add_case_weights(). workflow_set() will warn if any of the workflows specify an engine that does not support case weights—and ignore the case weights argument for those workflows—but will not fail.

Read more about case weights in the tidymodels at ?parsnip::case_weights.

Note

The package supplies two pre-generated workflow sets, two_class_set and chi_features_set, and associated sets of model fits two_class_res and chi_features_res.

The two_class_* objects are based on a binary classification problem using the two_class_dat data from the modeldata package. The six models utilize either a bare formula or a basic recipe utilizing recipes::step_YeoJohnson() as a preprocessor, and a decision tree, logistic regression, or MARS model specification. See ?two_class_set for source code.

The chi_features_* objects are based on a regression problem using the Chicago data from the modeldata package. Each of the three models utilize a linear regression model specification, with three different recipes of varying complexity. The objects are meant to approximate the sequence of models built in Section 1.3 of Kuhn and Johnson (2019). See ?chi_features_set for source code.

See Also

workflow_map(), comment_add(), option_add(), as_workflow_set()

Examples

```
library(workflowsets)
library(workflows)
library(modeldata)
library(recipes)
library(parsnip)
library(dplyr)
library(rsample)
library(tune)
library(yardstick)

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

data(cells)
cells <- cells %>% dplyr::select(-case)
```

```
set.seed(1)
val_set <- validation_split(cells)

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

basic_recipe <-
  recipe(class ~ ., data = cells) %>%
  step_YeoJohnson(all_predictors()) %>%
  step_normalize(all_predictors())

pca_recipe <-
  basic_recipe %>%
  step_pca(all_predictors(), num_comp = tune())

ss_recipe <-
  basic_recipe %>%
  step_spatialsign(all_predictors())

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

knn_mod <-
  nearest_neighbor(neighbors = tune(), weight_func = tune()) %>%
  set_engine("kknn") %>%
  set_mode("classification")

lr_mod <-
  logistic_reg() %>%
  set_engine("glm")

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

preproc <- list(none = basic_recipe, pca = pca_recipe, sp_sign = ss_recipe)
models <- list(knn = knn_mod, logistic = lr_mod)

cell_set <- workflow_set(preproc, models, cross = TRUE)
cell_set

# ------------------------------------------------------------------------------
# Using variables and formulas

# Select predictors by their names
channels <- paste0("ch_", 1:4)
preproc <- purrr::map(channels, ~ workflow_variables(class, c(contains(!!.x))))
names(preproc) <- channels
preproc$everything <- class ~ .
preproc

cell_set_by_group <- workflow_set(preproc, models["logistic"])
cell_set_by_group
```

Index